

UNIVERSITY OF NEWCASTLE UPON TYNE

DEPARTMENT OF COMPUTING SCIENCE



Constructing Highly-Available Distributed Metainformation Systems

Alcides Calsavara

Ph.D. Thesis

MAY 22, 1996

NEWCASTLE UNIVERSITY LIBRARY

095 52009 5

Thesis L5648

**PAGINATED
BLANK PAGES
ARE SCANNED AS
FOUND IN
ORIGINAL
THESIS**

**NO
INFORMATION
MISSING**

**TEXT BOUND CLOSE TO THE SPINE IN
THE ORIGINAL THESIS**

ORIGINAL COPY TIGHTLY BOUND

TO MY SON PEDRO

TO MY WIFE ESTEL

TO MY PARENTS IRINEU AND ELVIRA

Abstract

This thesis demonstrates the adequacy of an object-oriented approach to the construction of distributed metainformation systems: systems that facilitate information use by maintaining some information about the information.

Computer systems are increasingly being used to store information objects and make them accessible via network. This access, however, still relies on an adequate metainformation system: there must be an effective means of specifying relevant information objects. Moreover, distribution requires the metainformation system to cope well with intermittent availability of network resources.

Typical metainformation systems developed to date permit information objects to be specified by expressing knowledge about their syntactic properties, such as keywords. Within this approach, however, query results are potentially too large to be transmitted, stored and treated, at reasonable cost and time. Users are therefore finding it difficult to navigate their way through the masses of information available.

In contrast, this thesis is based on the principle that a metainformation system is more effective if it permits information objects to be specified according to their semantic properties, and that this helps managing, filtering and navigating information. Of particular interest is object orientation because it is the state-of-the-art approach to both the representation of information semantics and the

design of reliable systems.

The thesis presents the design and implementation of a programming toolkit for the construction of metainformation systems, where information objects can be any entity that contains information, the notion of views permits organising the information space, transactional access is employed to obtain consistency, and replication is employed to obtain high availability and scalability.

Keywords: metainformation, metadata, information discovery, information retrieval, object query, object-oriented database.

Acknowledgments

I am very grateful to my supervisor, Professor Santosh Shrivastava, who gave me support and encouragement throughout all my years of work. Our discussions were fundamental for the development of this research.

I would like to thank all the members of the Arjuna team as they have been very supportive throughout the development of Stabilis, particularly Graham D. Parrington, Stuart M. Wheeler, Mark C. Little and Steve J. Caughey. Special thanks go to Luiz E. Buzato, who was my partner in the development of most of this research. Many staff members of the Department of Computing Science deserve my gratitude. In particular, I wish to acknowledge Ms Shirley Craig, who helped me so much in obtaining bibliographical references for this research.

The list of friends that gave me inspiration and technical support to carry on this work is quite long. In particular, I am very grateful to Fernando Capretz, Raymundo Macêdo, Robert Burnett, Francisco Brasileiro, Sergio Cavalcante, Abelardo Montenegro, Suzana Montenegro and Marinho Barcellos.

Financial support for this research has been provided by Conselho Nacional de Pesquisa e Desenvolvimento (CNPq, Brazil), grant number 201905/91.4. I wish to thank Nelson Prugner for his very efficient administrative support.

Finally, I wish to express my gratitude to Estel, my wife, who always stayed with me in those difficult moments of a long research work.

Contents

Chapter 1	Introduction	1
------------------	---------------------	----------

1.1	Motivation	3
1.2	Objective	5
1.3	Overview	6
1.4	Outline	9

Chapter 2	Related Work	13
------------------	---------------------	-----------

2.1	Naming Systems	14
2.2	File Systems	18
2.3	Resource Discovery Systems	21
2.4	Distributed Systems	26
2.5	Database Systems	35
2.6	Other Approaches to Information Systems	39
2.7	Conclusions	42

Chapter 3	Object Engine Architecture	43
------------------	-----------------------------------	-----------

3.1	Object Engine Structure	43
3.2	Object Engine Operation	51
3.3	Conclusions	57

Chapter 4 Object Model Concepts 59

4.1 Object	61
4.2 Class	62
4.3 Object Relationship	65
4.4 Class Relationship	70
4.5 Inheritance	73
4.6 Summary	78
4.7 Conclusions	81

Chapter 5 Class Space Organisation 83

5.1 Overview	85
5.2 Schema Definition	88
5.3 Database Definition	99
5.4 Conclusions	102

Chapter 6 Meta-object Model 103

6.1 Meta-schema	106
6.2 Meta-object Mapping	112
6.3 Summary	124
6.4 Conclusions	127

Chapter 7 Object Space Organisation 129

7.1	Indices	129
7.2	Views	132
7.3	Contexts	133
7.4	Conclusions	134

Chapter 8 Stabilis Toolkit 137

8.1	Implementational Components	138
8.2	User Object Manipulation	157
8.3	Query Language	162
8.4	Query Resolution	167
8.5	Object Engine Set-up	172
8.6	Application Development	173
8.7	Performance	177
8.8	Conclusions	178

Chapter 9 Conclusions 181

9.1	Thesis Summary	181
9.2	Thesis Contributions	182
9.3	Evolution of Ideas and Experiments	184
9.4	Future Work	186

Appendix A	Object Model Definition	191
	A.1 Values, Types and Domains	192
	A.2 Attribute	196
	A.3 Relationship Elements	200
	A.4 Class Elements	210
	A.5 Method	216
	A.6 Class	222
	A.7 Object	226
	A.8 Relationship	233
	A.9 Single Inheritance	244
	A.10Class Hierarchy	248
	A.11Class Conformity	250
	A.12Partial Ordering of Classes	252
Appendix B	Theorem Proof	255
Appendix C	Meta-classes Definition	259
Appendix D	Meta-object Mapping	267
Appendix E	Components Definition	281
	E.1 Indices	281
	E.2 Views	284
	E.3 Contexts	285
Appendix F	Query Language Syntax	289
Bibliography	293

List of Figures

1.1	Object engine functional overview	11
2.1	The Arjuna class hierarchy.	31
3.1	Object engine structural architecture	44
3.2	Typical configuration of object engines	52
3.3	Effects of a sequence of object engine operations	58
4.1	General structure of an object	63
4.2	Example of related objects	63
4.3	Class diagram notation	64
4.4	Example of class diagram	65
4.5	Graphic notation for related objects	67
4.6	Example of associated objects	67
4.7	Example of tightly aggregated objects	69
4.8	Example of loosely aggregated objects	69
4.9	Example of tightly and loosely aggregated objects	70

4.10	Example of graphic notation for relationships	72
4.11	Graphic notation for class derivation	74
4.12	Simple class hierarchy	74
4.13	Simple schema for bibliographical references	79
5.1	Graph notation	89
5.2	Example of references between classes	91
5.3	Example of root-subtree	93
5.4	Example of schema aggregation	96
5.5	Relative self-containment of schemas	101
6.1	The meta-schema	108
6.2	Example schema for meta-object mapping	113
6.3	Example schema with annotated elements	125
6.4	Meta-objects for schema in Figure 6.3	126
6.5	Sets of mapping meta-objects in Figure 6.4	128
7.1	Classes with instances for the definition of indices	131
7.2	Example of context with views	135
8.1	Implementational components of object engines	139

8.2	Typical structure of Arjuna user classes' methods	149
8.3	Typical structure of managed classes' methods	149
8.4	Transition diagram for object state	151
8.5	Attribute indexing implementation – a sample	153
8.6	Relationship indexing implementation – a sample	154
8.7	Class hierarchy for the tree representation of query expressions . .	169
8.8	A tree representation of a query	170
8.9	Example of a network resource: a BIBTEX file	174
8.10	Schema for bibliographical references	175
8.11	Graphical interface for bibliographical references	180
A.1	Tight aggregation between classes Car and Wheel	202
A.2	Associations between classes School and Person	204
A.3	Correspondence between related classes and related objects	205
A.4	Tight aggregation between instances of Car and Wheel	206
A.5	Tight aggregations between classes Door , Bolt and Window	209
A.6	Class hierarchy with extents	211
A.7	Example of method inheritance	220
A.8	Class hierarchy with labelled members	227
A.9	Correspondence between class and instance elements	230
A.10	Relationships between classes for a simple hypertext model	238
A.11	Complementary relationship variables in related objects	239
A.12	Example of inheritance arc	244
A.13	Example of graph notation	246
A.14	Example of graph \mathcal{G}	248
A.15	Specification enlargement and deep extent reduction in class path	252
D.1	Example schema with annotated formal elements	277

List of Tables

5.1	Example of schema aggregation	87
5.2	Example of self-contained set of classes	92
8.1	Query language operators precedence	167
8.2	Performance of the object engine for bibliographical references . .	178
A.1	Example of class name and superclass name elements in classes . .	212
A.2	Example of class name and class path elements in objects	213
A.3	Example of class extent and deep extent	215
A.4	Example of class specification	227
A.5	Example of object elements	231
A.6	Example of complementary relationship specifications	238

CHAPTER 1

Introduction

A *meta-information system* is a system that manages some information *about* an information base and normally delivers very specific services. Typically, a meta-information system is constructed to separate, in a single system, some data and activity which may be common to a set of systems that manipulate an information base; such systems are the clients of the meta-information system. Thus, meta-information systems may vary in purpose and approach and, accordingly, can be grouped in categories. For example, *data dictionaries*, frequently employed in database management systems, office automation systems and CASE tools, represent a category of meta-information systems for documenting information structure. As another example, *searching engines*, widely employed in global networks, represent a category of meta-information systems for resolving keyword-based queries formulated by clients who aim at discovering network resources.

Our thesis proposes a novel category of meta-information systems and investigates critical issues on constructing them. We introduce meta-information systems whose purpose is to provide an *object-oriented interface* to information contained in network resources in large-scale distributed environments. For simplicity, we call these meta-information systems *object engines*, to connote the similarity be-

tween them and searching engines; both object engines and searching engines map information extracted from network resources to references to these resources in order to resolve queries. More specifically, an object engine maintains information objects extracted from network resources: objects are instances of classes that model information contained in network resources. Thus, clients of an object engine benefit from it for they can express semantic knowledge (structure and relationships) about the target information, rather than simple syntactic knowledge expressed through keyword-based queries. In addition, an object engine may provide for preview of network resources (by examining object attributes extracted from network resources), navigation through objects (by traversing conceptual object relationships rather than actual links between network resources) and request of services pertaining to network resources (by calling object methods).

Constructing an object engine involves concepts and techniques usually found in database systems, information retrieval systems, distributed systems and programming languages. Basically, an object engine requires an object-oriented modelling technique, an object store (provided with concurrency control, remote access, replication and recovery), index management, class space management (schemas), object space management (views), a query language and operations for object manipulation (creation, modification, retrieval, navigation and deletion). While some of these subjects are well understood, others still need further development and, especially, the combination of all of them in a single system may prove to be a complex and challenging task. Our thesis defines a simple yet coherent and effective platform for constructing object engines, as a starting point for future development.

1.1 Motivation

Information systems are required to be more efficient, effective and reliable as the information base stored by and accessible via computers expands in quantity, diversity and distribution. The introduction of object engines within this context is motivated basically by the observation of the following facts:

1. *Searching engines may cause inefficiency in the cases where the target information has a structure.*

A critical factor in determining the efficiency of an information system is the average rate of relevant hits in query results; non-relevant hits represent waste of bandwidth, processor, storage and time. For this reason, it is important to have information systems that permit users to formulate queries where they express the maximum of their knowledge about the desired information, thereby contributing to increase the rate of relevant hits.

The searching engines typically available in global networks employ information retrieval techniques which permit users to formulate queries where they express knowledge about *syntactic* properties of information, basically by making use of keywords. For example, Archie [21] indexes keywords extracted from resource names, WAIS [30] supports full-content indexing and Harvest [7] supports summary-content indexing. Certainly, this approach is appropriate for the cases where either the information presents no structure or has a structure that is unknown to users. However, a problem that frequently arises in keyword-based information systems is the large size of results containing non-relevant hits.

A significant part of the network resources in the global information base has well-defined structure and well-defined interrelationship paths which, if properly exploited, would permit users to formulate queries where they express *semantic* knowledge about the target information. For example, every Standard General Mark-up Language (SGML) [60] document has an associated Document Type Definition (DTD) that contains a set of grammar rules specifying the document structure. In fact, it is reported in [15] that DTD permits representation of SGML documents as instances of an O_2 [19] database schema, thereby obtaining high-level query services. As another example, *structuring schemas*¹ are used in [1] for specifying a map between bibliographical references in `BIBTEX` [35] files to database elements.

2. *Object-oriented modelling provides for effective information representation.*

Object-oriented data modelling represents a current end-point in the evolution of data modelling and is advocated to be an effective approach for the representation of real-world complex entities and their relationships. This suggests that information contained in network resources, including the links between these resources, would perfectly be modelled using object-oriented concepts. Thus, a schema (a collection of classes organised in hierarchies), devised to represent the structure and the relationships of information contained in a collection of network resources, would allow users to formulate queries in a highly-structured fashion. Moreover, users would be able to navigate through information by navigating through objects, and perform operations on network resources by calling object methods.

¹A structuring schema consists of a grammar with semantic actions.

3. *Object orientation is widely employed in reliable distributed computing.*

Many distributed systems, including operating systems and platforms for distributed programming, advocate the use of object orientation as an adequate framework for their internal structuring and as a powerful abstraction at the user interface level. In particular, the object and action model of computation [37] is a widely accepted approach to reliable distributed computing. Examples of systems based on this model are Arjuna [62] and Camelot [13].

1.2 Objective

Our objective is to contribute towards the efficiency, effectiveness and reliability of information systems in large-scale distributed environments. We intend to accomplish this by introducing object engines, a novel category of meta-information systems, as a means to provide an object-oriented interface to information contained in network resources. Object engines are intended to be used in conjunction with searching engines traditionally employed for information discovery in global networks; whereas searching engines, which aim at ill-structured information, object engines aim at well-structured information. Thus, information systems would gain in efficiency through object engines because query results would present a higher rate of relevant hits, when compared with searching engines. The gain in effectiveness would come from the power of object-oriented data modelling; an object-oriented view of information would provide a propitious abstraction for developing applications to manipulate information that has complex structure and

relationships. Finally, the gain in reliability would be achieved due to the use of object-oriented techniques which are well-established in modern distributed systems; transactional access would provide for consistency, while object replication for high availability and scalability.²

1.3 Overview

Object engines present architectural and functional similarities with searching engines, as illustrated by the scenario depicted in Figure 1.1. For simplicity, we use the term *broker* to refer to both object engine and searching engine, to connote that both types of engine are intermediate agents between network resources and client programs. In general, client programs formulate queries which specify predicates that may yield true propositions for some entities of the information base. For this reason, the meta-information maintained by brokers to resolve queries should consist of a relation between copies of portions of the network resources and references to these resources; each copy should be mapped to a reference to the corresponding resource. While a searching engine maps keywords extracted from network resources to references to those resources, an object engine maintains objects extracted from network resources; each object should have references to the network resources from which it has been extracted. Thus, the *modus operandi*

²We say that an information system scales if the growth of the entire system does not cause exponential growth of (1) the information maintained by the individual components of the system, and (2) the performance of operations to enter, update, delete, navigate and search information.

of searching engines and object engines can be summarised as follows.

Searching engines:

1. Collector programs extract *keywords* from information resources and convey such keywords to searching engines.
2. Searching engines maintain indices that map keywords to information resources; they resolve keyword-based queries and return *references to information resources*. Searching engines may co-operate with each other in order to share index information.
3. Client programs formulate keyword-based queries and ask searching engines to resolve them. The resulting references are used for retrieving information resources.

Object engines:

1. Collector programs extract *objects* which are instances of classes (abstract data types) from network resources and convey such objects to object engines.
 2. Object engines maintain objects and indices for object attributes and relationships; they resolve object-oriented queries and return *objects*. Object engines may co-operate with each other in order to resolve queries.
 3. Client programs formulate object-oriented queries and ask object engines to resolve them. The resulting objects are used for the following purposes:
 - (a) retrieve information resources: objects may contain references to information resources
-

- (b) preview of information resources: objects may contain summaries of information resources
- (c) navigate to related objects: object engines resolve object references and return the resulting objects
- (d) perform operations on information: objects may provide methods which can manipulate the corresponding summaries and information resources
- (e) create and modify objects: in addition to objects extracted from information resources, object engines may maintain objects created by clients, thereby behaving as information resources as well

Therefore, collector programs are specific to each type of information resource and each type of broker; a collector program must understand the format or the interface of a network resource and must understand the interface of a broker. Also, client programs should understand the interface of a broker; depending on the type of broker, a client program may have a different spectrum of services and, by making use of these services, may perform specific tasks. On the other hand, a general architecture can be devised for each type of broker, in especial for object engines. For this reason, our thesis concentrates on the design and implementation of object engines with the purpose of defining a platform for constructing them and writing collector and client programs with simplicity, i.e., we make an effort to obtain a simple interface for object engines. Nevertheless, we also develop a specific collector program and a specific client program to demonstrate the validity of the devised design and interface.

1.4 Outline

Our thesis is structured as follows.

- Chapter 2** describes related work that we have built upon, including information discovery tools, object-oriented databases and object-based distributed systems.
- Chapter 3** presents an architecture for object engines that unites concepts of searching engines, object-oriented databases and distributed systems.
- Chapter 4** presents a simple object-oriented data modelling technique based on the most salient features of many recently proposed models.
- Chapter 5** presents a model for organising the class space based on conceptual hierarchies of schemas and formally defines *databases*.
- Chapter 6** presents a *meta-schema*, a collection of classes that model information about classes and schemas, and show how this information is mapped to *meta-objects*.
- Chapter 7** presents a model for indexing object attributes and relationships, and presents *views* and *contexts* for organising the object space.
- Chapter 8** describes an implementation of the platform for constructing object engines and illustrates its use through examples. The description of the implementation concentrates mainly on the management of object storage, including concurrency control, remote access (distribution), replication and recovery. The use of the system is dis-
-

cussed together with the description of a query language for object manipulation.

Chapter 9 provides conclusions and further research work.

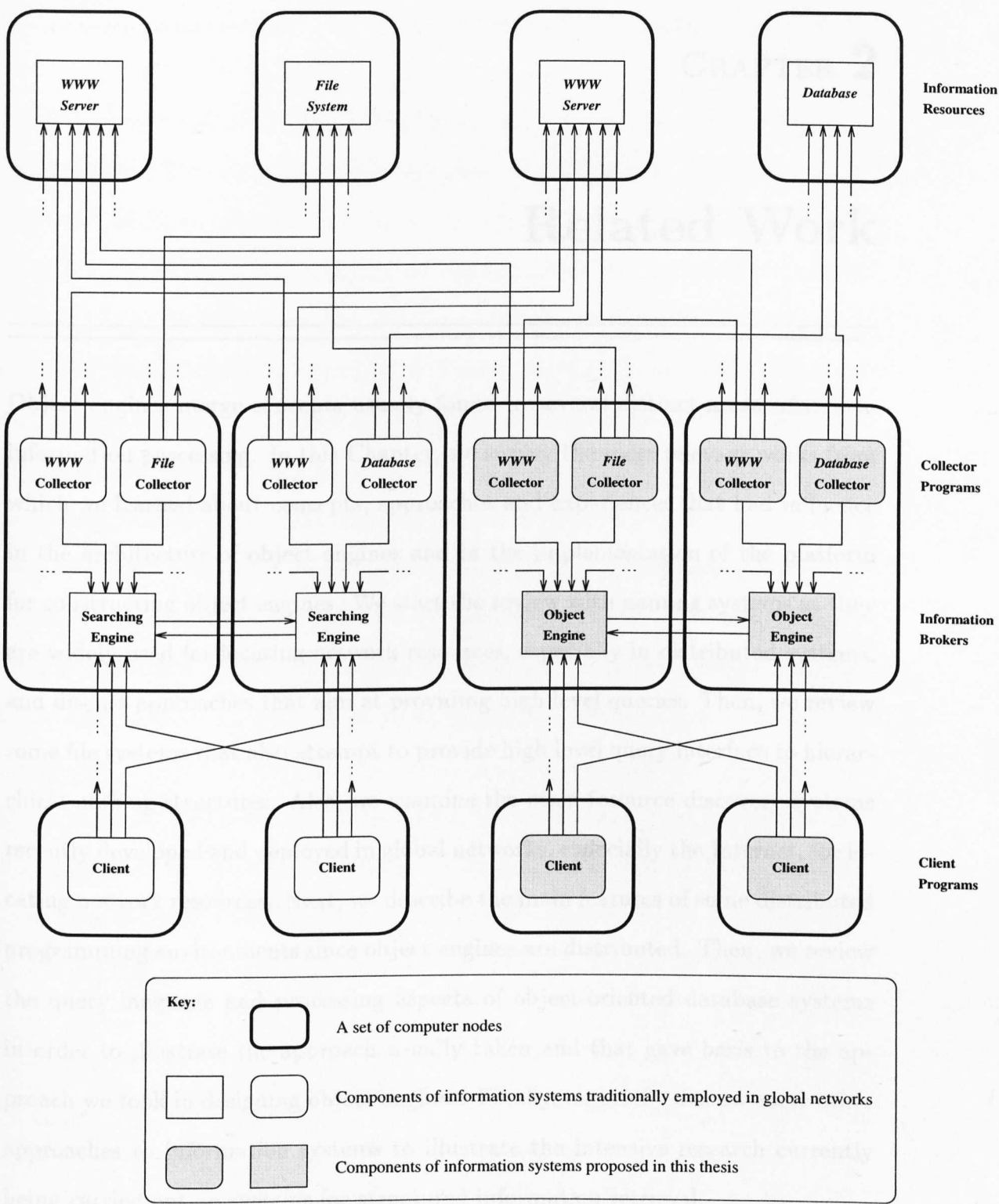


Figure 1.1 Object engine functional overview

CHAPTER 2

Related Work

Object engines merge concepts usually found in several distinct areas related to information processing. In this Chapter, we survey the most relevant works from which we learned about concepts, approaches and experiences that had influence in the architecture of object engines and in the implementation of the platform for constructing object engines. We start the review with naming systems as they are widely used for locating network resources, especially in distributed systems, and discuss approaches that aim at providing high level queries. Then, we review some file systems that also attempt to provide high level query interface to hierarchical naming structures. Also, we examine the main resource discovery systems recently developed and deployed in global networks, especially the Internet, for locating network resources. Next, we describe the main features of some distributed programming environments since object engines are distributed. Then, we review the query language and processing aspects of object-oriented database systems in order to illustrate the approach usually taken and that gave basis to the approach we took in designing object engines. Finally, we briefly discuss some other approaches to information systems to illustrate the intensive research currently being carried out on systems for structured information retrieval.

2.1 Naming Systems

Much of the research on naming systems in the recent years has been devoted to developing new models that avoid the restrictions imposed by the traditionally employed *hierarchical organisation of name spaces*. As discussed by D. B. Terry [65], hierarchical naming lacks expressiveness to represent the complexity of real-world entities, thereby preventing sophisticated queries. Universal naming systems, such as the Domain Naming System (DNS) [42, 43] and Lampson's Global Directory [36], generally restrict the structure of the name space to a hierarchy and support a simple search strategy in which a name denotes a path through the hierarchy.¹ This approach forces users to know enough attributes to construct a path name according to that partitioning; a naming tree cannot be searched with flat search requests, but rather must be traversed. The alternative models normally proposed are often called *descriptive naming* or *attribute-based naming*. A descriptive naming system accepts queries containing whatever information clients know about an object (not just its name) and responds with whatever information it possesses for each identified object (not just its address). Others argue that no single model can adequately address all situations. For example, Sechrest and McClennen [59] define a framework to blend the hierarchical and the attribute-based naming models. We review some naming systems by highlighting their aspects that influenced our work.

¹DNS basically maps host names into corresponding Internet addresses and vice versa.

Profile Naming Service

Profile [52] is an attribute-based naming system that provides *white-pages* service for large networks. The system takes as argument a set of attributes that describe an entity (a user or a organisation) and returns names of resources pertaining to that entity. Thus, Profile is a supplemental naming service; users consult Profile to learn names of resources that can be solved by existing naming systems. An attribute is a syntactic entity of the form *tag=value* that denotes a property or characteristic of an entity, such as *phone=3335890811*. A name server has a set of attribute tags defined for all entries. Such a set contains predefined tags and also may contain additional tags. Although attributes are tagged, clients are not required to include the tag when they query a name server. The reason is that tags are strings concatenated to attributes simply to enforce that attributes that denote different properties be syntactically unique; an attribute's tag can be determined from the syntax of its value.

Univers Attribute-based Name Server

Univers [9] is a generic attribute-based name server upon which naming services can be implemented. Univers consists of a light-weight relational database for storing information about resources, provided with a front-end interpreter and a server framework to support remote access. Conceptually, Univers maintains a database of objects, each of which corresponds to either some external resource that exists outside the name server (network resources) or some abstraction internal to the name server, such as types. Such a database allows clients to identify

objects with a set of attributes or properties that describe the object. Clients query Univers by submitting a *naming program* containing functions applied to sets of attributes. A naming program is constructed from a set of primitive operations that may be applied to the database, such as *select*, *project*, *create_object*, and others; a LISP-like programming language is provided for this purpose. Naming programs can either be submitted to the interpreter or they can be stored in the database as *function objects*. Univers imposes a type structure on the object database, thereby permitting users to isolate interesting classes of objects upon which they want to operate. The set of objects in the Univers database is partitioned in *contexts*, based on the authority that is responsible for administering the objects. Univers employs check-pointing and transaction logging to ensure database integrity and to facilitate failure recovery.

X.500 Directory Service

The X.500 Directory Service [68] has a semi-hierarchical naming scheme that blends the hierarchical and the attribute-based naming models. In this scheme, the attribute-based name space is restricted to a hierarchy: each level of the naming hierarchy contains a set of attributes, thereby supporting a unique *distinguished name* for each object. A X.500 name is composed of a sequence of comma-separated fields representing attribute-value assertions, where each assertion selects one node at a different level of the hierarchy. For example, the name $\langle C = \text{US}, O = \text{OSF}, CN = \text{Strauss} \rangle$ designates the object whose country is US, organisation is OSF and common name is **Strauss**. Users that have incomplete information about an object must traverse the hierarchy one node at a time, browsing

through attributes associated to objects, in order to learn the distinguished name of an object. For example, a user can start by selecting a subtree corresponding to a country, say US, then browse through all organisations within that country until eventually selects one of them, say OSF, and finally issue a *search* request having as an argument a string, say Strauss, which will be automatically associated with the corresponding tag in the current level of the hierarchy. The global name space is distributed among its participating sites. Administrative authority over portions of the global name space is delegated to different autonomous organisations, which can transfer authority over portions of their assigned subtrees. These portions are replicated on different servers. Each participating site maintains directory information about resources at that site, as well as administrative information needed for traversing the tree and maintaining proper distribution operation, including caching.

ANSAware Trading Service

The ANSA Naming Model [66] is a generic context-relative naming model that aims at interconnecting heterogeneous naming systems. An implementation of that model, the ANSAware Trading Service, basically consists of a two-level arrangement of naming systems: an *attributive naming system*, which relates attributive names to invocation names, and an *invocation naming system*, which relates invocation names to services. An *invocation name* unambiguously identifies a particular service and is used to interact with that service; the naming convention for invocation names is determined by the characteristics of the infrastructure (e.g., a socket name). An *attributive name* identifies a set of entities and

is attributed to an service by another entity; the naming convention for attributive names can exploit the *semantics* of the naming domain. The attributive naming system is further subdivided in two naming systems: a *type naming system* and a *property naming system*. The type naming system maintains a sub-typing directed acyclic graph of type names and an *is-an-instance-of* relationship between service instances and types. The property naming system, on the other hand, relates property names to property values. To reflect the *has-properties* relationship between service instances and a set of property name/value pairs, each property value is bound to an invocation name for a service instance. However, there is no association between types and properties.

2.2 File Systems

Similar to descriptive naming systems, some file systems provide an interface where files can be retrieved according to attributes rather than specifying a name structured according to a static naming tree. We review some of these file systems below.

Prospero File System

The Prospero File System [44] permits the building of large systems within which users construct their own virtual systems or *views* by selecting and organising files that they have identified as being of interest. Prospero relies on existing file systems for storage and supports multiple underlying access methods. Prospero is implemented as a distributed directory service that names individual files, plus

a file system interface that calls the appropriate access methods once a name has been resolved. The Prospero name space forms a directed graph in which intermediate nodes are directories, leaves are files, and edges are links which may have an attached filter program. By associating filters with links users can build customised views from existing ones, reorganising or extracting part of them. Thus, users can build views according to file attributes which are meaningful to them, independently of the physical organisation. Although Prospero permits files to be reorganised (and then designated) according to their attributes, it does not support attribute-based queries; users are forced to designate files according to some naming hierarchy.

Semantic File Systems

Semantic File Systems [22] provide flexible associative access to directories, files and portions of files into traditional tree-structured file systems by automatically extracting attributes from files and providing a query interface. Programs called *transducers* parse files according to the file types and generates file's entities (e.g., a procedure in a source code file or a single message in a mail directory) and their corresponding attributes. A query is a description of desired attributes of entities. Query resolution is performed through the use of *virtual directories* which are computed on demand, i.e., dynamically, to provide a user view of the file system contents. Virtual directories interact with existing file system facilities, and the syntax of a query is identical to file systems commands. For example, the query `ls -F /sfs/owner:/Smith` will return all files in `/sfs` that are owned by Smith. Thus, to have access to the contents of a file in that directory, the user should

simply use a standard file system command, such as `cat`, although the directory `/sfs/owner:/Smith` might not physically exist in the file system.

Nebula File System

The Nebula File System [8] merges the functionality of a traditional file system with information management operations provided by database systems. Nebula explicitly stores files as objects composed of a fixed set of attributes such as owner, protection, project, file type, and a special attribute called **text** to represent contents of files. The main purpose is to permit associative access to files using a combination of file attributes, i.e., a descriptive name or a query. Nebula file objects exist in a flat space of *contexts* and, within each context, users can organise files using a set of *views*, rather than directories. A view is the set of objects that are identified by a descriptive name, i.e., a view is dynamically created as the result of a query. Recursively, the resolution of a query is scoped within a view; the view obtained as the result of a query is an specialisation of the view against which the query was resolved. Since objects are registered with indices rather than directories, a view defines the portion of an index which must be considered for the resolution of a query. For example, the query `<format = text & project = plan2>` creates a view containing all files whose format is **text** and that pertains to the project **plan2**, within a given context. This view can then be used to scope the resolution of the query `<name = notes2.txt>` which will create a view containing the file object whose name is **notes2.txt**, within the scoping view.

Synopsis File System

The Synopsis File System [54] provides a logical interface to files through typed entities that summarise information corresponding to properties extracted from files in the form of a set of attributes added with a set of operations for interacting with the information content of a file. Accordingly, each such an entity is called *synopsis*. The type system uses an inheritance-based hierarchy to define types; a subtype inherits the attributes and the methods of its parent. A declarative language is provided for type definition and a scripting language is provided for specifying operations. A type repository provides persistent storage for type information. This allows types which were unknown at the time of compilation to be integrated into a client program in order to dynamically invoke operations on synopses. Although type information assists in extracting useful information from files, there is no report on the support of query facilities that make use of that information, such as attribute-based naming.

2.3 Resource Discovery Systems

Many tools for network resource discovery have appeared in the recent years with the increased availability of global networks. The basic function of these tools is to help users to locate information resources pertaining to a subject of interest. Typically users specify such subject through keywords and obtain as a result of the query references to network resources. Surveys on these and similar tools can be found in [58] and [46]. We review some of these tools to illustrate the different approaches typically taken.

Archie

Archie [21] is an index service for **ftp** sites that permits users to find files basically by specifying regular expressions (e.g., keywords) that match file names, thereby avoiding the difficulties caused by the hierarchical nature of Internet host names. Such index is solely built using file names which Archie obtains by regularly interacting via anonymous **ftp** with a collection of manually registered remote sites. Archie servers are replicated and the replicas maintained up-to-date through an efficient flooding-based algorithm. Archie indexes are very space efficient, but support limited queries; the success of Archie queries rely in file names reflecting their contents. Moreover, global flat indices tend to match too much information in query resolution as the information space grows.

WHOIS

WHOIS [27] is a centralised directory that collects distributed information about users, network numbers and domains. Users typically retrieve information about entities by specifying a keyword, such as the surname of a person. Each WHOIS server operates independently from each other and they are not linked together into a coherent directory system. Thus, a user may need to try different servers to find information and coordinate possible inconsistencies between them.

WAIS

Wide Area Information Servers (WAIS) [30] is a full-text information retrieval system consisting of a *directory of services* (a replicated global entity) and a col-

lection of databases that maintain complete inverted indexes on stored document contents. WAIS databases index documents in a wide variety of formats, and can be used to provide access to spreadsheets, databases, pictures, movies and sounds as well as text. Clients communicate with WAIS via an extension of the Z39.50 protocol [40] and are provided with relevance feedback. The return of a keyword-based query is a set of relevant descriptors that correspond to documents containing the keywords. These descriptors are ranked according to the frequency with which keywords are used, the proximity of keywords to each other, use of the keywords in the document title versus text, etc. Thus, the user is able to refine the query according to his interests. The interaction with WAIS is initiated through the directory of services which lets the user to select a set of databases to be queried.

Indie

Distributed Indexing or Indie [18] is a system for constructing cooperating brokers to index bibliographic data extracted from primary data sources (basically databases and other discovery tools) as well as from other brokers. Each broker is a database containing *object descriptors*, and such a database is described by a list of *generator objects*. An object descriptor is basically a record containing a number of attribute-value pairs corresponding to bibliographic data and other management data. The bibliographic data includes fields such as author, title and abstract, while the management data includes a field that identifies the network resource from which the bibliographical data was extracted (e.g., `host:caldera.usc.edu,port:32004`) and other fields to control the replica consistency

protocol used by Indie. Each generator object assigned to a broker contains a Boolean expression, such as (`keywords = network*`), that defines some information the broker should maintain. Such a Boolean expression is called *generator rule* to connote that a broker's database is generated and periodically updated by evaluating the rule over a number of other brokers. Thus, a broker must register its object generators with other brokers that index the corresponding information. If a broker *A* registers an object generator with a broker *B*, then *B* must periodically forward to *A* all object descriptors (creation and deletion) selected by the rule specified by the object generator. The interface with primary data sources is realised through specialised brokers called *gateways*. A gateway is supplied with raw information either by cooperative non-Indie servers or by directly collecting data from servers. The Indie architecture is completed by the *directory of services*, a replicated global entity that registers all brokers. Thus, users submit queries to the directory of services which returns a list of brokers whose object descriptors pertains to the user query. Then, in the second step, the user must rank the list of target brokers according to interest and submit the same query to them. As a final result of the query, the user will obtain a list of appropriate object descriptors which provides a basis for the user to retrieve the corresponding network resources through other retrieval systems.

Harvest

The Harvest system [7] consists of a set of tools for constructing systems that efficiently gather and index information extracted from network resources. In Harvest terminology, a *provider* denotes a server running standard Internet in-

formation services, such as FTP and HTTP. Thus, *gatherers* extract indexing information from *providers*, while *brokers* use that information to provide a query service. A *server registry* registers information about all gatherers and brokers, for the purpose of systems administration and also for users to look for an appropriate broker at search time.

Gatherers use the Essence system [25] to extract information from providers and compose *content summaries* corresponding to the network resources maintained by the providers. A content summary is composed of a number of attribute-value pairs, including a Uniform Resource Locator (URL) that globally identifies the corresponding network resource. Essence extracts relevant attributes from a network resource according to its format. For example, if the network resource is a mail repository then certain message header fields are extracted, or if the network resource contains bibliographic data then author names and titles are extracted. Essence supports a collection of approximately 25 common formats used in the Internet.

Brokers retrieve content summaries from gatherers and other brokers. (Content summaries are conveyed to brokers using an attribute-value stream protocol called Summary Object Interchange Format.) Brokers store these content summaries and generate corresponding index information. For that purpose, Harvest provides two index/search subsystems: Glimpse, which supports space-efficient full-text indexing, and Nebula, which supports attribute-based queries. Thus, clients submit queries containing a Boolean combination of keywords to brokers, obtaining as a result object descriptors constructed from the corresponding content summaries. Then, with the possession of these object descriptors, clients can

retrieve network resources.

Harvest topic-based brokers aim at coping with information overload and diversity to provide for scalability. Basically, there are two possible configurations for a gatherer to access a provider: either from across the network or running at the provider's site. The latter requires provider's site to run the Harvest software, but it is more efficient than the former because it causes less server load and less network traffic. Harvest uses replication of servers to enhance user-base scalability. For example, a server registry should be heavily replicated since it acts as a point of first contact for searches and system extension. Harvest adopts the data-conversion-and-migration approach rather than the query-translation-and-decomposition approach based on gateways (or filters) between information systems — a gateway may be a bottleneck and a source of communications delay, thereby compromising scalability. Also, Harvest uses object caching to reduce network load, server load, and response latency.

2.4 Distributed Systems

An object engine is an object-based distributed system. Thus, constructing an object engine requires an adequate distributed programming environment. Rather than providing an exhaustive review of the current environments, our intention is to illustrate the main features of some them, in particular the Arjuna system (described below) which we used as a basis for implementing the platform for constructing object engines. The increasing acceptance of the C++ programming language [64], especially in the area of system programming, has caused the emer-

gence of several environments providing support for programming parallel and distributed applications in that language. Following this trend, we give emphasis to such environments and highlight those features that most influenced in the architecture of object engines.

ANSA

The Advanced Networked Systems Architecture (ANSA) [4] describes the main principles of an environment for distributed systems development and corresponding run-time support with focus in distribution transparency but, at the same time, efficient exploitation of distribution. The architecture is not restricted to any particular programming language, operating system or network or hardware platform. On the contrary, the main goals of the architecture is the provision of interworking between autonomously managed networks and portability across a wide range of operating systems and programming languages.

An instance of the architecture, the ANSAware Testbench software [5], permits programmers to select the kinds of transparency required by applications. Basically, the transparencies supported are: *access transparency* (identical invocation semantics for both local and remote components), *location transparency*, *concurrency transparency*, *failure transparency*, *replication transparency* and *migration transparency*. The ANSA computational model defines the programming languages features that are necessary for this purpose, according to an object-oriented approach. All data is stored in distributed objects and accessed indirectly via interfaces; operations can only be invoked via their enclosing interface. In addition, it is possible for different objects to respond to the same operation,

possibly with different implementations. The ANSA computational model provides for both synchronous and asynchronous operations.

Distribution of client and server per distinct address spaces (process) demands an intermediate service to solve the initial addressing, i.e., to find a service when requested. In ANSA, this initial phase is called *trading*. A server invokes the *register operation* of the trading service in order to publish (export) all the operations (services) of one of its interfaces. Once exported, the services can be imported by any other client. Thus, a client imports an interface reference and then the client is able to invoke operations provided by the interface directly, without further participation of the trading service.

Basically, using ANSAware, the user writes a file in **IDL** (Interface Definition Language) with the definition of interface types and submit it to the stub compiler. Then a file corresponding to the server and another corresponding to the client is written in C with embedded **DPL** (Distributed Processing Language) commands to the ANSAware services (Trader, Notification, etc). The client and the server files are submitted to the C-preprocessor in order to convert the DPL commands to C commands. Then the C source codes obtained from the stub compiler and from the C-preprocessor are compiled and linked resulting in two executable codes: a server and a client.

Arjuna

Arjuna [62, 51] is a distributed transaction facility; it consists of a set of tools that supports the *object and action model of computation*, a widely accepted approach

to reliable distributed computing. In this model, programs consist of interacting objects which are instances of abstract data types, where every interaction happens within an atomic action, a programming abstraction that ensures serialisability, failure atomicity and permanence of effect.

serialisability Execution of concurrent programs are free from interference, i.e., it is equivalent to some serial execution.

failure atomicity A computation either commits, producing *all* the intended results, or aborts, producing no results. If any failure occurs, the appropriate use of backward error recovery undoes the results hitherto produced.

permanence of effect Any state change produced is recorded on *stable storage*, a type of storage that can survive system crashes with high change.

Coherence is accomplished by the enforcement of encapsulation; the state of the system is maintained solely by objects, and the state of each object is manipulated only by associated access methods, which, by definition, are the units of interaction. By ensuring that objects are recoverable and only manipulated within an atomic action, it can be guaranteed that the integrity of objects — and hence the integrity of the system — is maintained in the presence of failures such as node crash and message loss.

The main system facilities include object store (transparent persistence management), nested atomic actions (transparent distributed transaction management), remote object access (transparent remote method invocation using RPC), concurrence control, crash recovery and object replication. The object store provides access service to the passive state of persistent objects. The stable representation of an object (usually in disk) is machine independent in order to permit

its transmission between stable storage and volatile storage, and its transmission via RPC as well.

The model of concurrency control is shared variable (or object) with mutual exclusion and condition synchronisation through locking (there is an one-to-one correspondence between lock and object). The *strict two-phase locking* policy is adopted to ensure serialisability. Locks on objects are acquired within atomic actions (growing phase), and are released only when the outermost atomic action ends or aborts (instantaneous shrinking phase) [61]. There is no automatic detection of deadlock; applications should handle the situations where a lock request times out. Operations on objects are of type *read* or *write*, following the locking rule that permits *multiple reads, single write*.

It is assumed that the hardware components of the system are workstations (nodes), connected by a communications sub-system (for example, a local area network). A node is assumed to work either as specified or simply to stop working (crash). After a crash a node is repaired within a finite amount of time and made active again. A node is assumed to have both stable and non-stable (volatile) storage. All of the data stored on volatile storage is assumed to be lost when a crash occurs, while any data stored on stable storage remains unaffected by a crash. It is also assumed that faults in the communication sub-system are responsible for failures such as lost, duplicated or corrupted messages. The RPC system is assumed to be responsible for coping with such failures using well-known network protocol level techniques; it returns a failure exception to the caller if it suspects that the called server is not responding.

The current version of Arjuna is implemented as a standard C++ class library. Thus, it is tuned for the development of object-oriented applications. The Arjuna facilities are basically implemented by the class hierarchy depicted in Figure 2.1. Applications should define instances of the class `AtomicAction` and call its operations: `begin`, `end` and `abort`. The only objects controlled by atomic actions are those objects that are either instances of Arjuna classes or user-defined classes derived from the class `LockManager` – type inheritance is used to make user-defined classes members of the hierarchy shown in Figure 2.1.

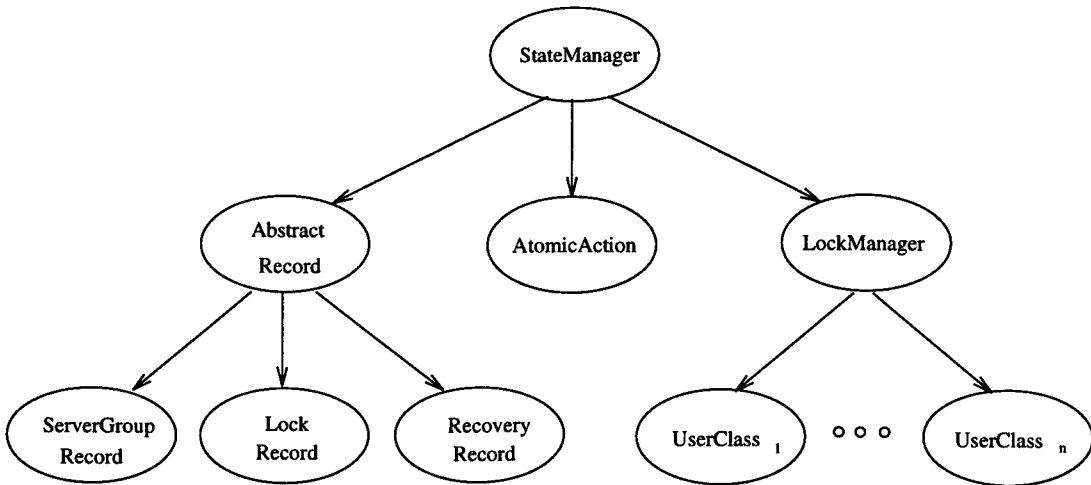


Figure 2.1 The Arjuna class hierarchy.

A tool called **Stub Generator** [50] processes definitions of C++ classes whose instances are persistent objects to be remotely accessed and, as a result, produces the corresponding client and server stub code. Transparency of location and access is obtained by making any invocation of an operation on the client stub object to trigger the same operation on the corresponding (remote) server stub object, using RPC.

Amber

The Amber system [14] aims at providing support for distribution and concurrency for C++ programs in homogeneous network of computers where each node is a shared-memory multiprocessor. Amber is based on a model of computation in which a collection of mobile objects distributed among nodes in a network interact through location-independent invocation (shared object abstraction). Thus, programs execute in a uniform network-wide object space, with memory coherence maintained at the object level. Objects are passive entities consisting of some data and a set of operations that can be invoked locally or remotely. The active entities in the system are thread objects; a typical application contains threads concurrently executing object operations on different processors. Programs are written in an object-based subset of C++, supplemented with primitives (*Start* and *Join*) for thread management and object mobility; threads invoking operations on an object move to the node where the object resides. The system is composed of a preprocessor to C++ and a runtime kernel which is linked with applications. Amber provides the programmer with a set of predefined object classes for managing threads, synchronisation and distribution. However, there is no support for persistent objects, primitives for reliable distributed computing or communication and cooperation between unrelated programs; Amber aims at concurrent programming on tightly-coupled machines.

DC++

The DC++ [57] system is an object-oriented extension of the OSF Distributed Computing Environment (DCE) [48] integrated with C++, without introducing any language modifications. The major features of the system are location independence and object mobility to circumvent certain deficiencies of the traditional client/server model supported by DCE. In the system, C++ objects are the basic units of distribution. All distributable objects are referenced using the DCE's universal unique identifiers (UUIDs), and the DCE Cell Directory Service (CDS) is used for optional retrieval of objects by name. Objects communicate by method invocations, independently of their location; remote invocations are mapped onto DCE remote procedure calls. A remote reference is implemented by a *proxy* indirection. A proxy contains a location hint for the referenced object and transparently forwards invocations based on DCE RPC; each node maintains a hash table for mapping the global identifiers within incoming invocations onto actual storage addresses of C++ objects. Mobility allows for modelling physical data transfer at a very high level of abstraction and also provides *explicit* control of distribution, such as when it is appropriate to co-locate communicating objects. Upon request by applications, objects can dynamically move between nodes. DC++ provides no support for object persistence or distributed transaction.

PANDA

The PANDA system [6] is a run-time package which supports distributed and parallel applications written in C++. The main system features are object persis-

tence, uniform global address space, user-level threads, object and thread mobility, and garbage collection. Distribution in PANDA is provided through object and thread mobility rather than using remote procedure calls (RPCs). According to the designers of PANDA the RPC mechanism makes it difficult to provide perfect distribution transparency in C++, such as handling pointers which may occur in the parameter list of a method. Object mobility is realised through a distributed shared memory (DSM) mechanism, assuming that the hardware platform consists of a network of homogeneous processors. The programming environment provides a primitive DSM which can be specified for any object creation. The basic mechanisms for thread management are provided by the system class **UserThread**. Applications classes derive from that class and implement a especial inherited method **code** which is automatically spawned when instances of the application classes are created. In addition, the class **UserThread** provides a method **migrate** that accepts as a parameter a destination node. The invocation of this method causes the thread be interrupted, transferred to the specified remote node and then resumed. For concurrency control, PANDA offers synchronisation objects such as semaphores and signals, thereby permitting to turn a class into a monitor. Persistence mechanisms are integrated into the run-time environment of the language and distributed transaction for persistent objects is supported. Applications use a primitive **persistent** for the classes whose instances should be persistent objects. Although PANDA does not extent the programming language, it requires applications to be instrumented with the primitives mentioned above; thus, applications need to be preprocessed.

2.5 Database Systems

Object engines and object-oriented database management systems have some common purposes: store objects and permit them to be queried against *schemas* that were previously devised for applications by applying a modelling technique (according to a certain object data model). Moreover, object-oriented database management systems normally share many features with distributed object-oriented systems since they both store objects. For example, as discussed in Section 2.4, transaction management, which is originally a feature of database systems, is also supported by some object-oriented distributed systems, such as the Arjuna system. In fact, the interest in object orientation has been a point of convergence of the work in several fields of research, and there seem to be a tendency to merging distributed systems, database systems and persistent programming systems under a single object-oriented framework. However, object engines are not intended to be full-fledged object-oriented database management systems. Our approach in providing a platform for constructing object engines is simply the extension of an distributed object-oriented transaction facility, namely the Arjuna system, with object query services. This approach is appropriate because, firstly, similarly to what happened with object-oriented distributed systems, the increasing acceptance of the C++ programming language has had strong influence on the design of object-oriented database systems in the recent years, and secondly the object data model of the Arjuna system is based on C++. For this reason, for our purposes, it suffices to review object-oriented database management systems solely with regard to query formulation and resolution, and in the context of C++. Although there are several systems that provide persistence through extensions to

C++, for conciseness, we will concentrate our review on just one of them, namely the ObjectStore database system [34, 49]. Other representative systems are Ontos [3] and ODE [2].

ObjectStore

ObjectStore [34, 49] is an object-oriented database system that supports persistence orthogonal to type, transaction management and associative queries, through an extended version of C++. The target applications are typically the ones that perform complex manipulations on large databases of objects with intricate structure, such as computer-aided software-engineering, computer-aided design and manufacturing, and geographic information systems. Such an intricate structure is normally realised by inter-object references, which must be traversed by associative queries in order to locate objects. ObjectStore is based on a client/server architecture: the client deals with objects while the server deals with *pages* only. For performance reasons, ObjectStore deals with the task of solving associative queries by moving database functionality into the client, rather than doing application-specific tasks on the server. The programming environment basically consists of a class library and a preprocessor. The library contains *collection* classes including sets, bags and lists. For example, a transient set **p** of instances of a class **person** can be created with the following C++ statement:

```
os_Set<person*> p;
```

Thus, class **person** may have data members **name**, **age**, **boss** (an object refer-

ence) and **children** (a multi-valued attribute) declared as:

```

string          name;

int             age;

person*         boss;

os_Set<person*> children;
```

And to make the set **p** persistent in a database **db**, the declaration would be:

```
persistent<db> os_Set<person*> p;
```

A query is a predicate surrounded by `[:]` specified by a *query operator*, typically over a single top-level collection. For example, the following query locates persons named **Amadeus** over the set **p**, and stores the result in a transient set **q**.

```
os_Set<person*>& q = p [ : name == "Amadeus" : ]
```

The selection predicate may be any C++ expression. For example, using the logical operator `&&` (and), the following query returns all persons whose age is between 10 and 20.

```
os_Set<person*>& q = p [ : age >= 10 && age <= 20 : ]
```

Object references are traversed using nested queries or the operator `->`, depending on the context. For example, the following query returns all persons who have a child whose name is **Zweig**.

```
os_Set<person*>& q = p [ : children [ : name == "Zweig" : ] : ]
```

And the following query returns all persons who have a boss whose name is **Verdi**.

```
os_Set(person*)& q = p [: boss-> name == "Verdi" :] :]
```

There is no concept of class extent in ObjectStore. Thus, indices can be defined for a collection (rather than for a class) in order to speed up query resolution. An index is firstly created and then associated to a collection. In general, predicates are over *paths*. Thus, for example, the following statements define indices for the paths **name** and **boss-> name** of class **person**, and associates them to collection **p**.

```
os_index_path name_path =pathof(person*, name)
os_index_path boss_name_path =pathof(person*, boss-> name)
p.add_index(name_path);
p.add_index(boss_name_path);
```

The maintenance of these indices are automatically done every time a program modifies the corresponding paths, such as every time that the data member **name** of an object of class **person** is modified. A keyword **indexable** should be added to the data member's declaration for this purpose. Thus, for example, the data member **name** should actually be declared as follows.

```
string name indexable;
```

ObjectStore provides a construct **foreach** to iterate over the members of a collection. For example, the following statement calls the function **print** for all persons in the set **p** who have children younger than 5.

```
foreach (person * x, p)
    print(x-> children[: age < 5 :])
```

Query resolution has four phases: analysis, code generation, strategy selection and execution. The analysis phase creates a parse tree representing the query. The code generation phase generates a set of functions to evaluate the query (when the generated code executes, the tree does not actually exist). The strategy phase notes (at run-time) the presence or absence of each index relevant to the query and propagates this information over the tree, selecting the appropriate functions to be executed. Finally, the execution phase executes the selected functions.

2.6 Other Approaches to Information Systems

Integration of Database and IR

Some systems integrate database technology with Information Retrieval (IR) techniques. Saxton and Rahavan [56] argue that while databases do not fulfill the requirements of today's information systems, such as unstructured decision making and weighted evaluations, IR systems cannot precisely select only and all relevant information. For this reason, they developed a system that augments a relational database management system with ranked queries, based on the Generalised Vector Space Model. That system also augments the relational model with the introduction of *generalisation* and *aggregation* for the designers found these concepts very useful in properly identifying views to be used in queries and, moreover, the representation of hierarchical structure of objects and relationships can be used during search, taking advantage of the semantics available. In another work, Harper and Walker [26] developed a system called ECLAIR that provides an interface for IR-type queries (basically using best-match retrieval techniques) on

top of the Ontos object-oriented database management system [3]. The database system is used for storing and indexing objects that represent the contents of network resources. However, ECLAIR does not integrate the query language provided by the database system with IR techniques. The main reason for using the database is to obtain concurrent access to data and reliable processing of data in the presence of system failures. On the other hand, Christophides et al. [15] uses the O₂ object-oriented database management system [20] to represent SGML [60] documents in order to benefit from recovery, concurrency control and high-level query services provided by the database system. For that purpose, they had to extend the O₂ query language in order to enable users to query data without exact knowledge of its structure, and using approximate match.

Document Databases

Some systems exploit the implicit structure found in text files to provide high level query and update facilities as exist in database systems. Examples of such files are electronic documents, programs, literature citations and mail messages. Loeffen [39] presents a survey containing a dozen of text models and systems. In general, these models describe texts by their structure, operations on the texts and constraints on both structure and operations. Their basic motivation is that normally retrieval systems deal only with two kinds of textual objects: the word and the document containing it, leaving unrepresented any intermediate structure. Bruza and van der Weide [10] define a stratified approach to hypertext systems. The authors argue that, these days, objects need no longer be modelled as amorphous things, especially, due to emerging standards such as the Standard General Mark-

up Language (SGML) [60] and the Office Document Architecture (ODA) [47]. In fact, Christophides et al. [15] map the type information present in the prologue of each SGML document – the Document Type Information (DTD) – in an O_2 schema. In another work, Consens and Milo [16] show how word indexing and region indexing can be combined with extended database query optimisation to provide efficient access to semi-structured textual information. Basically, they use the PAT text indexing system and translate high-level database queries on files to expressions in the PAT algebra. Järvelin and Niemi [29] introduce a declarative query language that allows data aggregation simultaneously with complex data restructuring without the user having to describe, explicitly, how the result documents are constructed from the available ones. The motivation for this approach is that there is no static hierarchical structure among subdocuments in which all users would always want their result documents.

Metadata Systems

There has been an intensive research on metadata systems for different purposes. Hsu et al. [28] describe a metadata system for general information resources management in heterogeneous, distributed environments, in order to integrate computerised enterprises. The system extends the traditional approach to metadata taken by data dictionaries with the inclusion of knowledge resources such as business rules, control for sequential interactions and global decision processes for parallel systems interactions. They employ a method called Two Stage Entity Relationship which, in addition to structured data representation as relations, permits the representation of semantics through a functional model in the form of

production rules. Madsen et al. [41] developed a metadata system whose purpose is to locate relevant information giving some information about it, the user and the context of the query, also by incorporating semantics to resource metadata. This is achieved through a model for locating data given the type of data required and details of the context in which it is to be used. Grosky et al. [24] developed a metadata system called Content-Based Hypermedia for browsing structured media objects, i.e., portions of images, videos and audios. The structure and relationships of these objects are represented through an object-oriented schema. Thus, users can browse through this meta-information space to discover properties and relationships between media objects.

2.7 Conclusions

There is an intensive research in information systems to provide means for locating relevant network resources, and many approaches exist to make use of information semantics. However, to our knowledge, there is no proposal that exploits object orientation at the user interface. Full-fledged object-oriented database systems, on the other hand, permit sophisticated queries but they are normally too resource intensive to be deployed in global networks. Distributed systems, especially transaction-based systems, offer a sounding basis for developing robust information retrieval systems in large-scale environments but they lack query facilities. Therefore, object engines are proposed to consolidate appropriate features of these proposals and systems, thereby providing an efficient, effective and reliable structured information retrieval service.

CHAPTER 3

Object Engine Architecture

An architecture for object engines must integrate most of the fundamental features of searching engines, object-oriented databases and distributed systems. In this Chapter we describe such an architecture by outlining a set of components, their interconnections and interaction with client programs; we give a more detailed description of each component in subsequent Chapters. The architecture is built upon the assumption that there exists a system that supports the object and action model of computation for reliable distributed computing.

3.1 Object Engine Structure

The physical components of an object engine include objects, indices, meta-objects, views and a context, as illustrated in Figure 3.1. Additionally, the meta-objects encompasses other conceptual components, namely classes, schemas, meta-classes and a meta-schema. All physical components of an object engine may be replicated and distributed over any number of nodes of a distributed system. These conceptual and physical components, including their interconnections (represented by arrows in the Figure), are explained as follows.

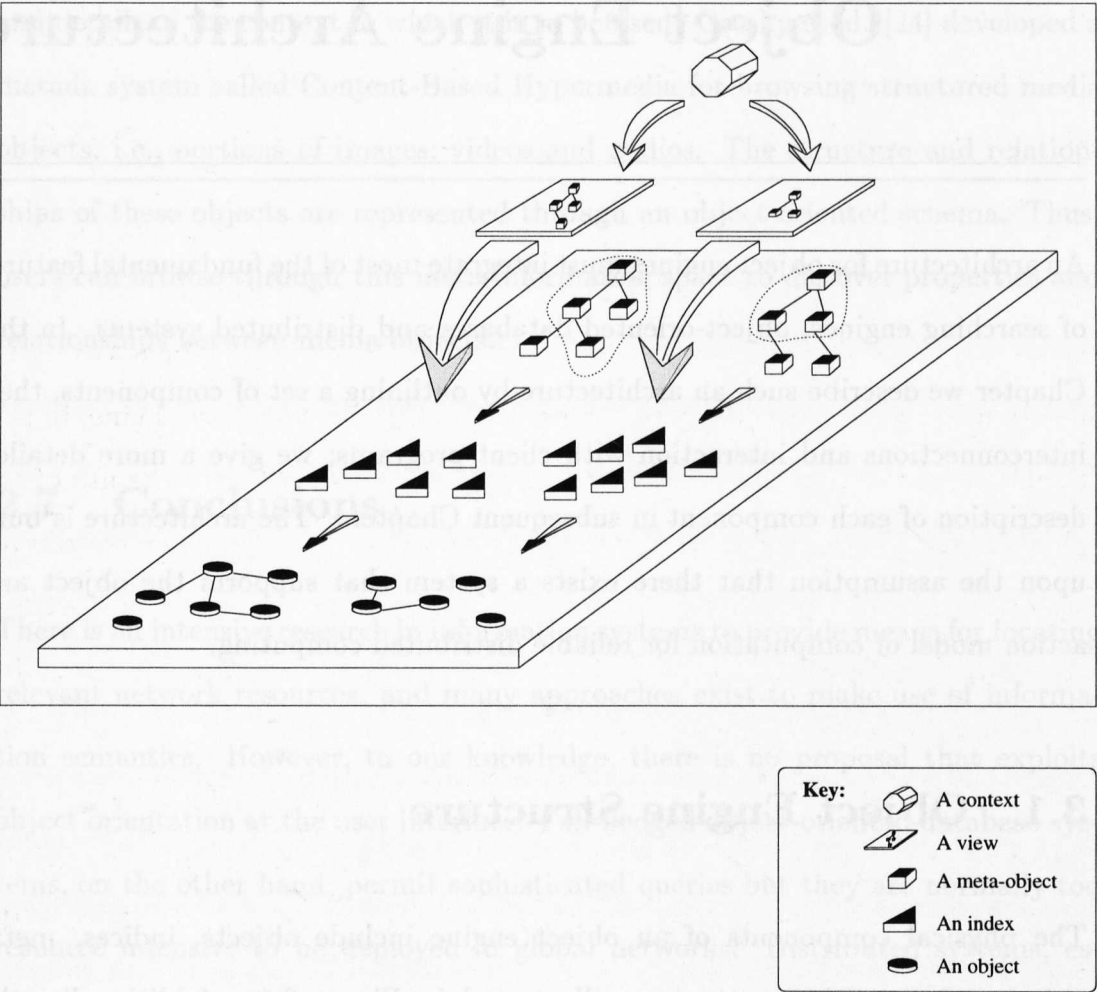


Figure 3.1 Object engine structural architecture

3.1.1 Basic Components

The primary function of an object engine is to maintain objects extracted from network resources: objects must correspond to pieces of information contained in network resources, and objects must be related to each other according to the relationships between those pieces of information. Additionally, in conformance with the object-oriented model, an object must provide a set of operations (i.e., methods) to encapsulate its state.

Typically, object-oriented systems define objects as instances of abstract data types (i.e., classes), as this provides for both a strongly-typed system and a means to formulate queries according to a conceptual *schema*. An object engine, in particular, benefits from those both features: strong typing permits the use of the object and action model of computation in order to obtain a reliable distributed system, while queries formulated according to schemas permit users to express their knowledge about the target information in a highly-structured fashion, similarly to object-oriented database systems. For this reason, information contained in network resources must be modelled by schemas composed of classes that represent the structure of the information, and objects extracted from network resources must be instances of such classes. In summary, an object engine has the following basic components:

Class : an abstract data type that defines a set of attributes, a set of relationships with other classes and a set of methods.

Object : an instance of a class. An object is atomic, and it is a unit of concurrency, replication and caching. We call the set of all objects maintained by

an object engine *object base*.

Schema : a set of classes that model some information. Indirectly, a schema designates a set of objects: the set of all instances of the classes that belong to the schema.

3.1.2 Meta-data Components

Although classes and schemas could be simply regarded as conceptual components,¹ an object engine must maintain a physical representation for them due to the following reasons:

1. *Documentation*. Information about classes helps users to navigate through information: users learn what categories of information exist, how information is structured, and, consequently, how to formulate good queries. Moreover, this information is useful for developing tools on top of an object engine, similarly to the use of *meta-data* by third-party vendors for developing tools on top of database systems [33].
2. *Query Resolution*. Information about classes permits a query interpreter to analyse query expressions and resolve queries.
3. *Software Management*. Information about classes permits automatic generation of code.
4. *Administration*. Systems administrators need to record classes and schemas definitions in order to update and re-use them.

¹Objects are components which implicitly require a physical representation.

We refer to the physical representation of information about classes and schemas as *meta-data* to comply with the nomenclature normally used in database systems.² Thus, meta-data must be maintained by object engines and made accessible to client programs, similarly to objects extracted from network resources. Ideally, for the sake of homogeneity, all client programs should interact with object engines through a single interface, independently of whether a client program manipulates objects or meta-data. Hence, meta-data should be represented as ordinary objects. Moreover, a single approach to representing information extracted from network resources and meta-data would permit all information maintained by object engines to be manipulated in the same fashion by users and administrators, including that meta-data also would be distributed and highly available.

Therefore, we define a special schema to represent classes, attributes, relationships, methods and schemas; the objects which are instances of the classes in this schema comprise the meta-data maintained by an object engine. Accordingly, we use the prefix “meta” to designate the meta-data components of an object engine, as follows.

Meta-schema : a special schema that models meta-data.

Meta-class : a class that belongs to the meta-schema.

Meta-object : an object that is an instance of a meta-class. We call the set of all meta-objects that represent all schemas defined for an object engine *meta-object base*. Since a meta-object is an object, for a given object engine, the object base is a superset of the meta-object base.

²Another suitable name would be *data dictionary*.

3.1.3 Index Components

Query resolution is presumed to be the most requested service of object engines. For this reason, efficient algorithms and appropriate data structures for query resolution are essential for obtaining high performance in object engines. We call the components of object engines used for this purpose *indices* to comply with the nomenclature used in database systems and searching engines.

Basically, an object query is a predicate expressed in terms of classes, attributes and relationships represented in a certain schema. For example, a query may be formulated to retrieve an object of class **Writer** that is related to an object of class **Story** whose attribute **title** has value equal to **The Picture of Dorian Gray**. Roughly, the resolution of this query needs firstly to map the attribute value **The Picture of Dorian Gray** to an object of class **Story**, and secondly map that object to an object of class **Writer**. Hence, an object engine should maintain the following index components:

Attribute Index : a relation between attribute values and object references; an attribute value is mapped to an object reference when the corresponding object has an attribute with that value. An attribute index has references to all objects which are instances of the class to which the attribute belongs.

Relationship Index : a relation between object references and object references; an object reference is mapped to another object reference when both corresponding objects are connected to each other. A relationship index has references to all pairs of related objects which are instances of two particular related classes.

The association between an index and its corresponding attribute or relationship is physically represented using meta-objects:

- An attribute index is referred by the meta-object that represents the corresponding attribute.
- A relationship index is referred by the meta-object that represents the corresponding relationship between two classes.

Thus, a query can be resolved by navigating through meta-objects until the necessary indices are identified. For example, to retrieve objects of class **Person** whose attribute **age** has value greater than 18, firstly the meta-object that represents the class **Person** must be retrieved, and next the meta-object that represents the attribute **age** must be retrieved, thereby obtaining a reference to the index that contains references to the target objects.

3.1.4 Organisational Components

An object engine may contain a large number of schemas, which may designate a large number of classes and a large number of objects. Furthermore, these schemas may model information about a wide range of topics, and an object engine may have a large number of users, distributed over a wide area network. Hence, an object engine must organise its schemas in a fashion that permits the following features:

1. *Security*: users may be allowed a restricted access to the object base.
 2. *Customisation*: users may have interest only in part of the object base.
-

3. *Efficiency*: users may wish to scope query resolution to a particular schema.
4. *Scalability*: type space administration must be decentralised in large-scale distributed environments.

In general, an object engine must permit part of a schema to be designated as another schema (i.e., a sub-schema), and it must permit schemas to be grouped to form a larger schema (i.e., a super-schema). Thus, an object engine may contain a number of conceptual hierarchies of schemas, possibly with intersections between them. Moreover, a schema does not necessarily have to be available to users; a schema may be created only for the purpose of deriving other schemas (sub-schemas and super-schemas) from it. For this reason, we differentiate the schemas which are available to users by calling them *views*, to connote that they define how information maintained by object engines is actually observed by users and administrators. In particular, administrators should be provided with a special view that corresponds to the meta-schema. Accordingly, we call such a special view *meta-view*.

Since schemas are represented by meta-objects, a view is simply defined by selecting a certain set of meta-objects. Conceptually, a view corresponds to a schema, a set of indices and a set of objects. Physically, a view should be either a simple reference to a set of meta-objects or it should be a full representation of that set of meta-objects. For practical reasons, we decided for the latter approach: a compact representation of meta-objects permits faster query resolution, and it is a suitable unit of concurrency, replication and caching. Also for practical reasons, an object engine must maintain a directory of all views. We call such a directory *context*, to connote that it defines an independent name space. Thus, a program

initiates interaction with an object engine by retrieving the context that designates the object engine and, next, the program asks the context to return specific *views*. In summary, an object engine has the following organisational components:

View : an entity containing meta-data equivalent to a schema defined for the object engine; a view is derived from the meta-objects that represent a schema and, accordingly, contains references to the corresponding indices.

Context : a global entity containing references to all views defined for the object engine; an object engine is designated by a context.

3.2 Object Engine Operation

An object engine must be set up by a bootstrap program to create a context, a set of meta-objects to represent the meta-schema, all corresponding indices and a meta-view. Thereafter, an administrator program, using the meta-view, can create other meta-objects and corresponding indices to represent other schemas, and then create other views. Thus, using these views, client programs can create, update, delete, retrieve and traverse relationships between objects. Figure 3.2 shows a typical configuration of object engines and programs. The object engine is distributed over the nodes of a single local area network (LAN). The programs are classified according to their specific purpose and relative location (either in the same LAN where the object engine is located or in a remote LAN) and described as follows.

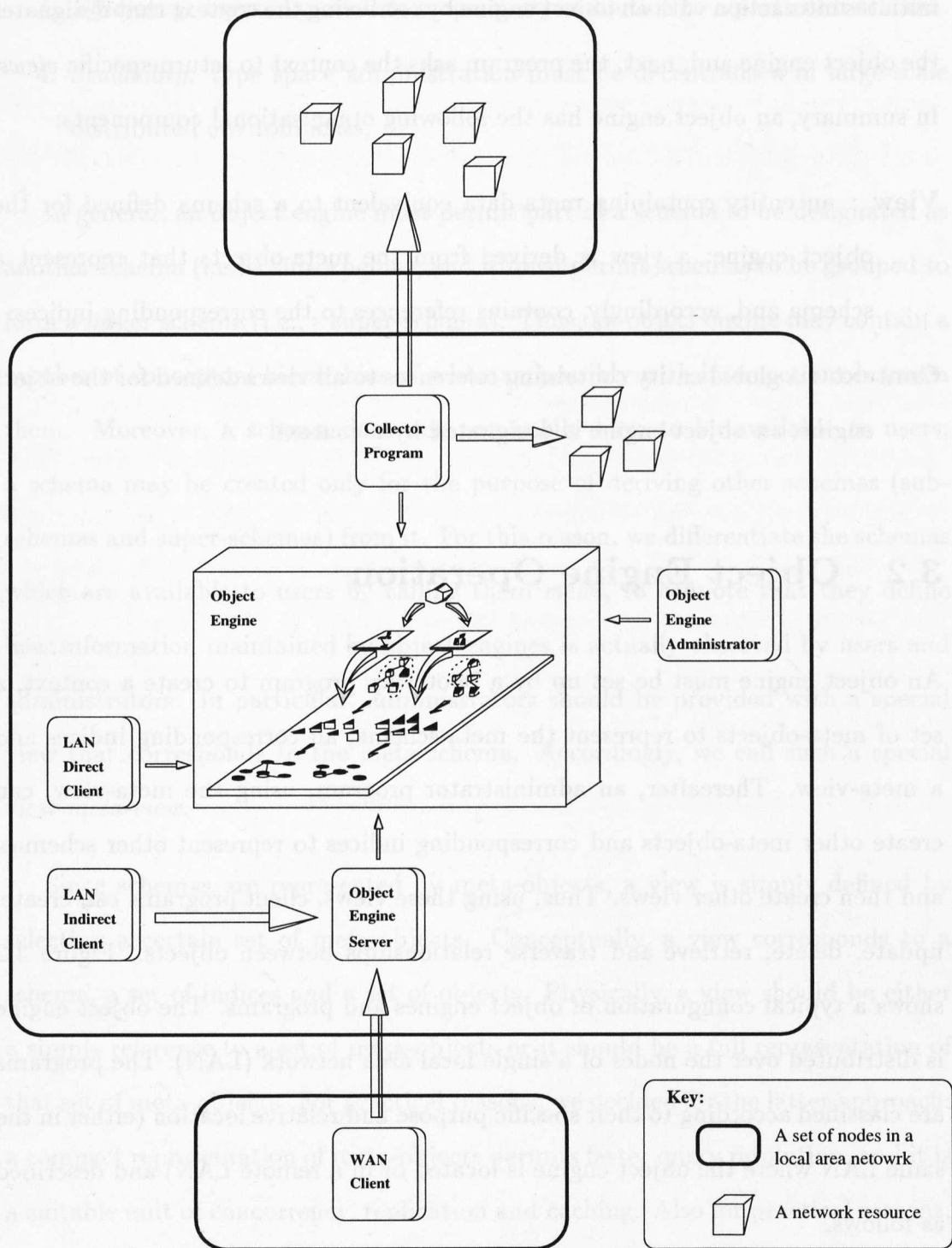


Figure 3.2 Typical configuration of object engines

3.2.1 Collector Program

A collector program periodically extracts information from network resources and updates the object base maintained by an object engine. A collector program must understand the structure or the interface of a network resource and, at the same time, it must understand a schema that models the information contained in the network resource. Thus, a collector program is able to translate information from its “natural” representation to objects. Roughly, the normal operation of a collector program should consist of the following steps:

1. Extract from network resources the pieces of information pertaining to object attributes and relationships, according to a schema that models the information contained in the resources.
2. Assemble pieces of information pertaining to object attributes in tuples that conform to classes specified in the schema; each tuple should form a valid object state.
3. Use object states for creating, modifying and removing objects from the object base.
4. Use pieces of information pertaining to object relationships for connecting and disconnecting objects that already exist in the object base.

Typically, a collector program should run in the same LAN where the object engine is located, while the network resources manipulated by a collector program may be located anywhere in the global network.

3.2.2 LAN Direct Client

A LAN direct client is a program that is located in the same LAN where an object engine is located, and that directly manipulates the object base maintained by the object engine. A LAN direct client can be of one of the two following types:

1. *Schema-specific application*: the client can manipulate the part of the object base designated by a specific schema.
2. *General query interpreter*: the client can manipulate the whole object base.

We discuss LAN Indirect Client in Section 3.2.4.

3.2.3 Object Engine Administrator

An object engine administrator is a program that manipulates information about schemas, i.e., it manipulates the meta-object base. Since the meta-object base is a subset of the object base, an object engine administrator can be simply regarded as a particular case of LAN direct client: it is a client program that manipulates the objects designated by the meta-schema. And another consequence is that a general query interpreter also can be used as an object engine administrator.

3.2.4 Object Engine Server and Clients

An object engine server is a program that is located in the same LAN where an object engine is located, and that provides a set of operations for manipulating the object base maintained by the object engine; the server is an intermediate

between a client program and the object engine. Actually, an object engine server is identical to a LAN direct client, except that it accepts calls from other programs: the main purpose of an object engine server is to permit a client program located in a given LAN to have access to an object engine located in another LAN; in this case the client program is called **WAN client**, to connote that it operates over a wide area network. Naturally, a client program located in the same LAN where an object engine is located also can have access to the object engine through an object engine server; in this case the client program is called **LAN indirect client**.

Similarly to a LAN direct client, an object engine server can be of one of the following types:

1. *Schema-specific server*: the server manipulates the part of the object base designated by a specific schema. In this case, the client is necessarily specific to the schema, i.e., it is a schema-specific application that has access to an object engine located in a distinct LAN.
2. *General query server*: the server can manipulate the whole object base. In this case, the client can be either a schema-specific application or a general query interpreter that has access to an object engine located in a distinct LAN.

In both cases, an object engine server may completely conceal the object engine from the client by providing an appropriate set of operations. Furthermore, a server can provide an interface that is not even object oriented, thereby dispensing with the need of the knowledge of schemas by clients. For example, a server could

provide an interface for simple keyword-based search; in this case the client could be an information browser traditionally used in global networks.

Finally, a client of an object engine server may be another object engine server, thereby being possible to configure networks of object engines. This arrangement would permit object engines to co-operate.

3.2.5 Summary

Let us summarise object engine operations with a simple example. Let us consider that firstly an object engine is set up, secondly an object engine administrator creates a schema named **Eg**, thirdly a collector program creates objects of classes in the schema **Eg** and, finally, a LAN client program manipulates these objects. The effects of this sequence of operations are illustrated in Figure 3.3 and explained as follows.

1. *Bootstrap*. The set up of the object engine encompasses the following steps:

- (a) create a context
- (b) create meta-objects to represent the meta-schema
- (c) create indices that correspond to the meta-schema
- (d) index the meta-objects that represent the meta-schema
- (e) create the meta-view

2. *Administrator*. The creation of the schema named **Eg** consists in:

- (a) create meta-objects to represent the schema **Eg**
-

- (b) create indices that correspond to the schema *Eg*
- (c) create a view for the schema *Eg*

We should note that the object engine administrator makes use of the meta-view. Also, we should note that the meta-objects that represent the schema *Eg* can be automatically indexed (by the indices for the meta-schema) when they are created since, at that stage, no longer there are circular dependencies.

3. *Collector*. The collector program makes use of the view *Eg* to create objects of the classes designated by the schema *Eg*.
4. *Client*. The LAN direct client also makes use of the view *Eg* to manipulate objects of the classes designated by the schema *Eg*.

3.3 Conclusions

The object engine architecture integrates features of searching engines, object-oriented databases and distributed systems in a homogeneous fashion. Some of the most salient features of the architecture are the following:

- All components of an object engine are objects with transactional access, including concurrency control, and that can be replicated and distributed: hence highly available.
 - A uniform representation of objects extracted from network resources and meta-data permits all programs to interact with object engines through a
-

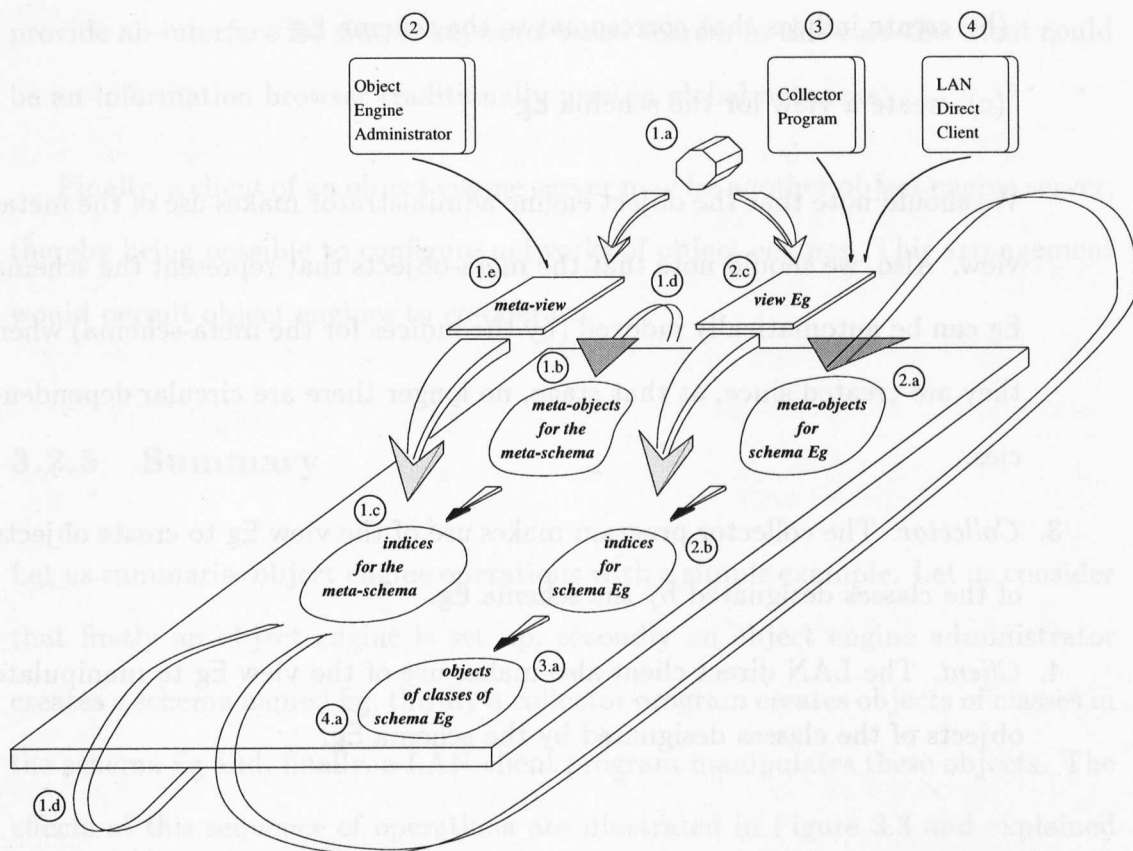


Figure 3.3 Effects of a sequence of object engine operations

single interface, and makes it simple for users to learn about schemas and formulate queries.

- The use of meta-objects and general indices to represent object attributes and relationships provides an effective data structure for query resolution.
- A combination of schemas, views and contexts provides an effective means of organising the information space in large-scale distributed environments.

The remaining Chapters of our thesis describes the components of the architecture in more detail and discusses implementation issues. Accounting, object placement (i.e., clustering) and migration will not be discussed.

CHAPTER 4

Object Model Concepts

Information objects are modelled by employing the notions of encapsulation, identity, classification, inheritance (generalisation/specialisation) and relationship found in object-oriented programming and in database systems. The purpose of modelling objects is to define a *schema*¹, a collection of classes which describe the properties of the objects and are arranged in a certain way to ensure that the objects which belong to these classes compose a consistent database and, therefore, can be properly manipulated.

Although object-oriented modelling concepts have been employed in several domains of applications since they were first introduced by the designers of Simula [17] and currently represent an end-point in the evolution of data models, in this Chapter we delineate such concepts for the following reasons:

1. There is not a single standard on these concepts. Thus, we present an interpretation of object orientation that complies with a standard *de facto* which has been established with, firstly, the widespread use of the programming language C++ throughout considerably large part of academia and

¹Also referred to as *object data model* in the literature.

industry and, secondly, with the adoption of the OMG CORBA [45] by the industry as an architecture for inter-platform cooperation.

2. Although most of the object models proposed in the literature and/or commercially available support the notion of relationships between objects, they normally treat such a relationship as a simple “pointer” from one object to another, thereby not making a clear distinction between the possible different semantics of these pointers, namely associations and aggregations. On the other hand, some object models, especially the ones developed as a “natural” evolution of the relational model, make such a distinction. An example is the Object Modeling Technique (OMT) [55] where associations and aggregations are modelled differently. Another example is the ORION database management system [32] which provides the notion of *composite objects* to permit the modelling of “part-of” relationships between objects. For this reason, we show the importance of modelling object relationships with proper semantics through examples and go further by introducing a new concept, namely *loose aggregation*, explained in Section 4.3, to distinguish the cases where objects are only *conceptually* from the case where they are *physically* part of other objects.
3. We introduce a graphic notation together with the concepts which is used throughout the rest of this thesis to represent schemas diagrammatically.
4. We present examples of application of the concepts to illustrate their adequacy to our purposes.

While the concepts are here described informally as this suffices for the self-containment of this thesis, a formal definition of the object model is presented

in Appendix A, where the set and the graph theories are employed to prove the correctness of the model. Also, we present many other examples of applications to give support to our point of view about the adequacy of the model.

4.1 Object

An object is defined by an **identity**, a **state** and an **interface**, as illustrated in Figure 4.1.

- The identity is a unique identification that permits the object to be referred to unambiguously. It is represented by a unique **name**.
- The state is a structure containing properties of the object. It is represented by **attributes** (values of primary types, such as string and integer) and **relationships** (references to objects).
- The interface contains the operations which can be applied to the object. Similarly to an extent of an abstract data type, the interface *encapsulates* the state. It is represented by **methods** that have exclusive access to the state.

Figure 4.2 illustrates a simple example of two related objects: the object named **X** represents a client of a bank, and the object named **Y** represents the client's account. The client has an attribute **name** of string type, a relationship with the account, and the methods **update_name** and **check_balance** which, respectively, have parameters of string and integer types. The account has the attributes **number** and **balance** of integer type, a relationship with the client, and the methods

deposit, withdraw and check_balance, each of them with a parameter of integer type. The dashed lines in the Figure illustrate the relationships between the objects.

4.2 Class

A **class** is an abstract data type that defines an object state structure and the corresponding interface. Basically, a class consists of:

1. a set of attribute specifications
2. a set of relationship specifications
3. a set of method specifications

An **instance** of a class is an object whose state consists of the set of attributes and relationships specified by the class, and whose interface contains the set of methods specified by the class. Thus, a class stands for a set of objects that have *some* common structure and behaviour.

Attributes and methods are components of just one class, while a relationship is generally between two classes. For this reason, we discuss relationships separately in Section 4.3 and Section 4.4.

- An attribute is specified by a name, a primary type and a Boolean value that indicates whether the attribute is *key*, i.e., whether the attribute can be used in queries.
-

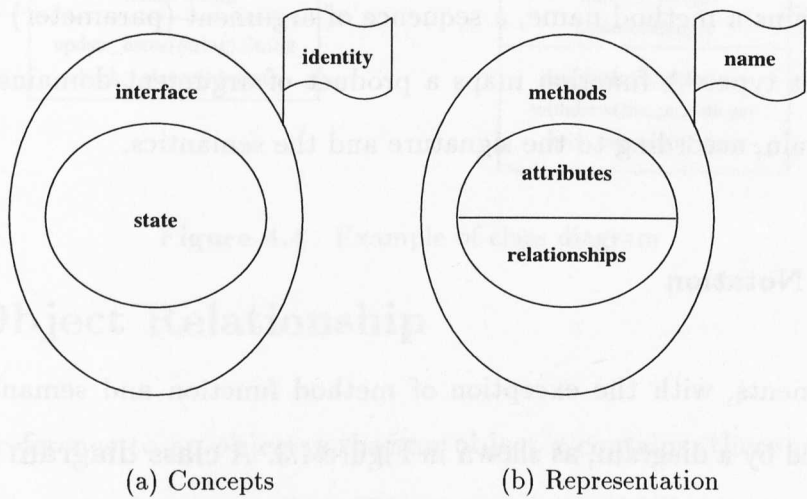


Figure 4.1 General structure of an object

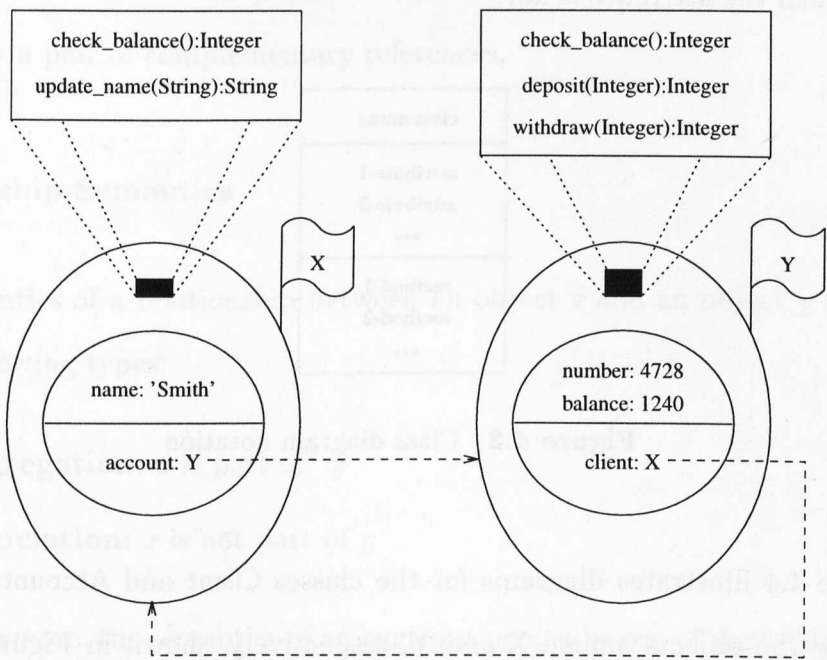


Figure 4.2 Example of related objects

- A method is specified by a signature, a function and a semantics. A signature contains a method name, a sequence of argument (parameter) types and a result type. A function maps a product of argument domains to a result domain, according to the signature and the semantics.

Graphic Notation

Class elements, with the exception of method function and semantics, can be represented by a diagram, as shown in Figure 4.3. A **class diagram** is composed of up to three stacked rectangles: the top rectangle contains the class name, the intermediary rectangle contains attribute specifications, and the bottom rectangle, if present, contains method signatures. An attribute name is preceded by a star symbol when the attribute is key.

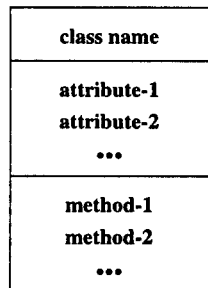


Figure 4.3 Class diagram notation

Figure 4.4 illustrates diagrams for the classes **Client** and **Account**, which are classes for the objects named X and Y, respectively, shown in Figure 4.2. The attribute **name** of class **Client** and the attribute **number** of class **Account** are key attributes.

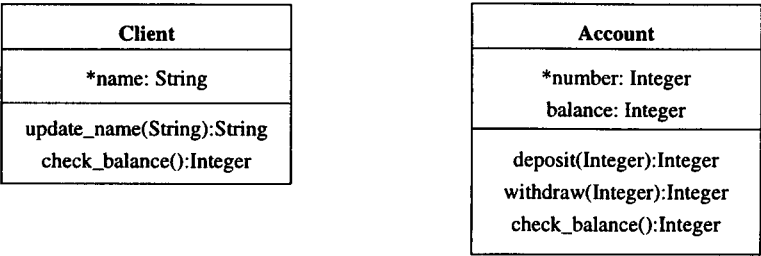


Figure 4.4 Example of class diagram

4.3 Object Relationship

For every reference to an object y that an object x contains, there is a complementary (inverse) reference to x in y . The term *relationship between x and y* refers to a pair of complementary references, and the term *relationships of x* refers to all relationships between x and any other object. Also, for simplicity, we say that a relationship between two objects is *bi-directional* to denote that a relationship is defined by a pair of complementary references.

Relationship Semantics

The semantics of a relationship between an object x and an object y can be one of the following types:

- 1. **Aggregation:** x is part of² y
- 2. **Association:** x is not part of y

Furthermore, the semantics of an aggregation can be one of the following types:

²Aggregations correspond to the so-called *complex objects*.

1. **Tight Aggregation:** x is physically part of y
2. **Loose Aggregation:** x is conceptually part of y

In both types of aggregation, x is a **component** of y , and y is an **aggregate** containing x .

Graphic Representation

A set of related objects can be represented as a graph where vertices correspond to objects and edges correspond to relationships. In such a graph, two meanings can be assigned to edges:

1. **Navigational:** An edge corresponds to an object reference and, consequently, it is directed: an edge from object x to object y means that x contains a reference to y .
2. **Semantical:** An edge corresponding to an aggregation is directed from the aggregate to the component, while an edge corresponding to an association is not directed.

We refer to a graph where the edges have a navigational meaning as a **navigational graph**, and a graph where the edges have a semantical meaning as a **semantical graph**. A navigational graph is useful to show paths that can be traversed. A semantical graph, on the other hand, is useful to show relationship semantics. However, since relationships are always bi-directional, a semantical graph implicitly represents a navigational graph and, consequently, it is sufficient for both purposes. Therefore, we preferentially use semantical graphs and we

use navigational graphs only when relationship semantics is not relevant. Figure 4.5 shows the convention we will use henceforth for the graphic representation of related objects.

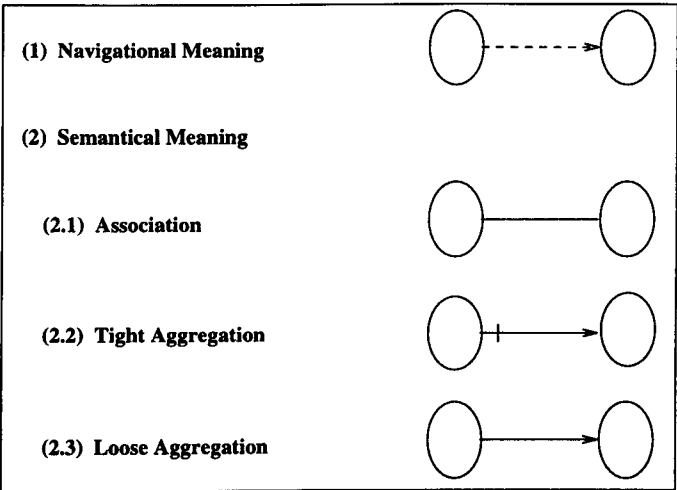


Figure 4.5 Graphic notation for related objects

Associated Objects

An object can be associated to any number of objects. Consequently, the semantical graph for a set of associated objects is a generic graph. An example of association is the relationship between the objects client and account discussed in Section 4.1. Figure 4.6 illustrates the corresponding semantical graph.

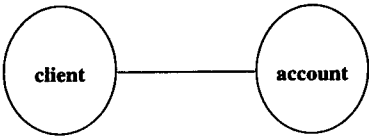


Figure 4.6 Example of associated objects

Tightly Aggregated Objects

An object can be physically part of at most one object. Consequently, the semantical graph for a set of tightly aggregated objects is a directed tree, i.e., a single hierarchy where objects which are relatively higher in the hierarchy contain objects which are relatively lower in the hierarchy. For example, the aggregation hierarchy in Figure 4.7 represents a journal composed of five articles which are composed of certain numbers of pages.

Loosely Aggregated Objects

An object can be conceptually part of any number of objects. Consequently, the semantical graph for a set of loosely aggregated objects is a directed acyclic graph, more specifically, a collection of intersecting hierarchies. For example, the aggregation graph in Figure 4.8 represents a grouping of persons in sports clubs, where a person can be member of more than one club.

Tightly and Loosely Aggregated Objects

An object that is physically part of another object also can be conceptually part of other objects. For example, the articles contained in a number of journals can be grouped in logical folders according to subject, as shown in Figure 4.9.

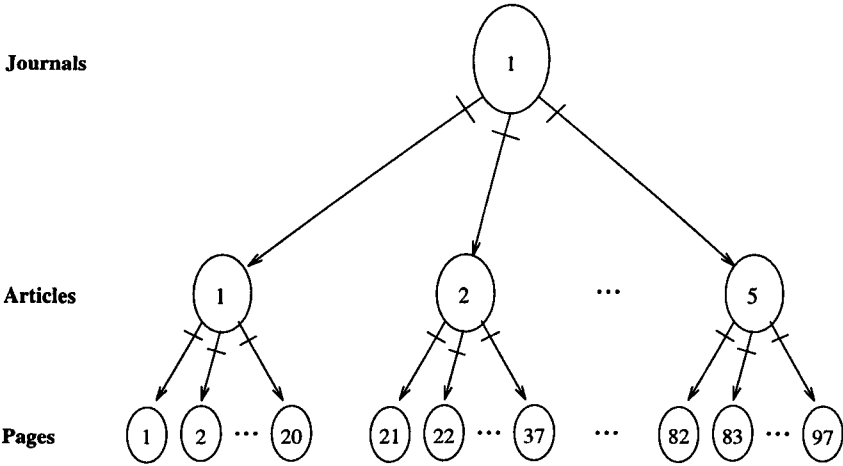


Figure 4.7 Example of tightly aggregated objects

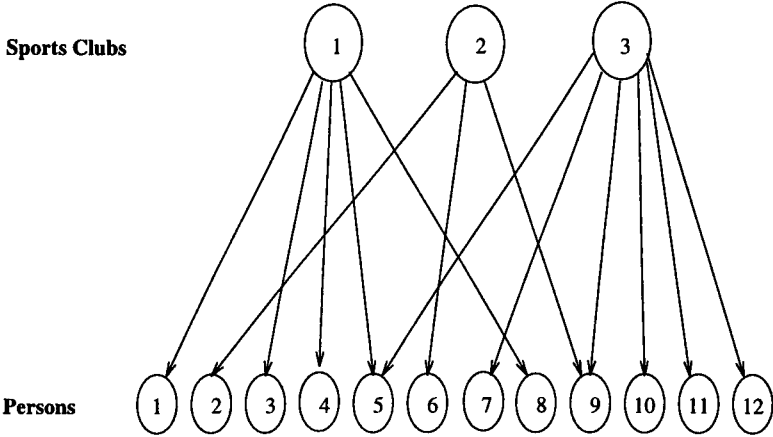


Figure 4.8 Example of loosely aggregated objects

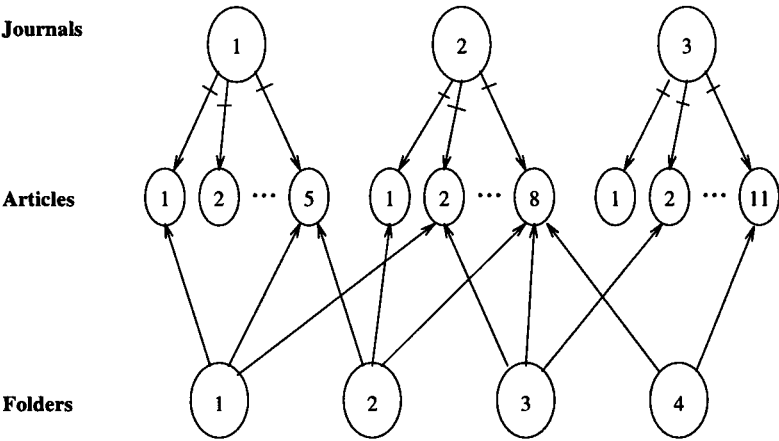


Figure 4.9 Example of tightly and loosely aggregated objects

4.4 Class Relationship

The permitted relationships between objects are specified through relationships between classes. A relationship between a class X and a class Y defines how an object x that is an instance of X can be related to an object y that is an instance of Y , by specifying the following items:

1. **Semantics:** It defines whether the relationship between x and y is an association, a tight aggregation or a loose aggregation. In both cases of aggregation, either X is the aggregate class and Y is the component class or vice-versa. Then, in the relationship between x and y , the instance of the aggregate class is the aggregate object and the instance of the component class is the component object.
2. **Role:** A label is assigned to X and another label is assigned to Y in order to define the roles of x and y , respectively, in their relationship. If the role of a class is not explicit then the class name is assumed as the class role. Roles are useful as both documentation and a measure for unambiguous

identification of relationships.

3. **Multiplicity:** A pair of integer values is assigned to X and another pair is assigned to Y in order to define the minimum and the maximum number of instances of X that can be related to a single instance of Y and vice-versa. Because these values specify the lower bound and the upper bound of a set of object references they are referred to as **minimum cardinality** and **maximum cardinality**. In the case of tight aggregation semantics in particular, the multiplicity of the aggregate class is constant: 0 as the minimum cardinality and 1 as the maximum cardinality.
4. **Key:** A Boolean value is assigned to X and another Boolean value is assigned to Y to indicate whether the relationship is *key* with respect to each class, i.e., whether the relationship can be traversed in queries using path expressions. If **true** is assigned to X , for example, then the relationship can be traversed from Y to X .

Graphic Notation

A relationship between two classes is graphically represented by either a line or an arrow between the corresponding class diagrams, and labelled with the role and the multiplicity of each class, as illustrated by the examples in Figure 4.10.

- An association is represented by a dotted line, a loose aggregation is represented by a dotted arrow and a tight aggregation is represented by a dashed arrow. In both cases of aggregation, the arrow is from the aggregate class to the component class.
-

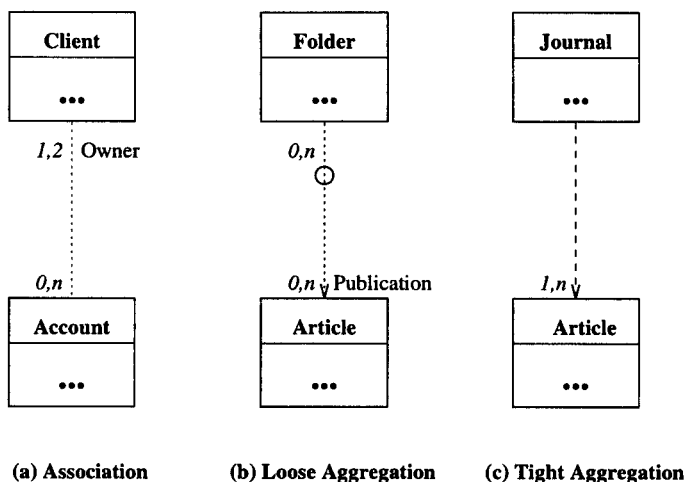


Figure 4.10 Example of graphic notation for relationships

- A label and a pair of integer values on each end of the line or arrow, respectively, denote the role and the multiplicity of the class represented by the nearest attached diagram. The multiplicity of the aggregate class in a tight aggregation is not shown since it is constant.
- The role of a class can be omitted when the class name is the role. The roles shown in Figure 4.10 is **Owner** for class **Client** and **Publication** for class **Article** when related with class **Folder**.
- A circle on the line that represents a relationship means that the nearest class is not key in the relationship. For example, **Folder** is not key in the relationship with **Article**. As a consequence, a path expression from **Article** to **Folder** cannot be specified.

4.5 Inheritance

A class *Y* can be derived from a class *X* in order to augment and/or modify the set of specifications pertaining to *X*, according to the following rules:

1. *Y* can have additional attributes, relationships and methods.
2. *Y* can substitute (overload) method functions and semantics.

The set of specifications pertaining to *Y* is a superset of the set of specifications pertaining to *X*; it contains the attributes, relationships and methods defined by *X* and *Y*. Consequently, an instance of *Y* consists of the attributes, relationships and methods defined by *X* and *Y*. For this reason, we say that *X* and *Y* have an inheritance relation, more specifically, *Y inherits from X*.

Graphic Notation

The graphic notation to represent that a class named *Y* is derived from a class named *X* is an arrow from the class diagram for *X* to the class diagram for *Y*, as shown in Figure 4.11.

Class Hierarchy

A class hierarchy, also denominated inheritance hierarchy, is defined by a set of classes and their inheritance relations. Any class hierarchy can be extended by deriving new classes from its classes. As shown in Figure 4.12, for example, a class named *Z* is derived from *Y* and a class named *W* is derived from *X* to extend the class hierarchy defined by *X* and *Y*.

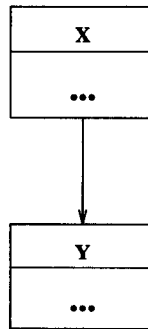


Figure 4.11 Graphic notation for class derivation

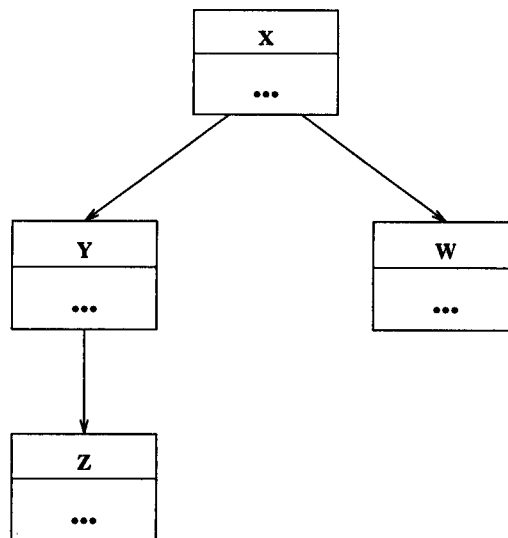


Figure 4.12 Simple class hierarchy

Inheritance Invariants

To ensure the inheritance relation properties and for the sake of simplicity of the object model, in a class hierarchy, the following invariants with respect to inheritance must hold:

1. No class inherits from itself (neither directly nor indirectly).
 2. Every class inherits directly from at most one class (single inheritance).
-

According to these invariants, by representing classes and direct inheritance relations as a graph where classes correspond to vertices and direct inheritance relations correspond to arcs, we have that such a graph is a *directed tree*. For this reason, a direct inheritance relation is referred to as an **inheritance arc**.

Inheritance Relation Properties

The inheritance relation is neither reflexive (a class cannot inherit from itself) nor antisymmetric but it is transitive. For example, the fact that Z inherits from Y and that Y inherits from X implies that Z inherits from X. Thus, we differentiate between direct and indirect inheritance relations: X and Y have a direct inheritance relation, while X and Z have an indirect inheritance relation.

Class Instance and Extent

We differentiate between direct and indirect instances. A **direct instance** of a class α consists of the attributes, relationships and methods defined and inherited by α . An **indirect instance** of α , on the other hand, consists of the attributes, relationships and methods defined and inherited by a class that (directly or indirectly) inherits from α .

Moreover, the **extent** of α is the set of all direct instances of α . The **deep extent** of α is the union of the extent of α and the extents of all classes that inherit from α . For example, the deep extent of Y includes the extents of Y and Z. As another example, the deep extent of X includes the extents of X, Y, Z and W.

Class Conformity and Instance Substitutability

Since classes are abstract data types, we say that a class α *conforms to* a class β if α contains at least the specifications pertaining to β . The conformity relation is useful to determine whether an object can be used in a certain context. If it is specified that a direct instance of a class α is expected (in assignments and as parameter in method invocations, for example) then a direct instance of any class that conforms to α is acceptable. Thus, an instance of a class can be *substituted* by instances of different classes, i.e., by objects of different forms (*polymorphism*).

Obviously, a class α conforms to itself (reflexivity) and, from the definition of inheritance, we have that if a class α inherits from a class β then α conforms to β . Therefore, a direct instance of a class α can be substituted by a direct instance of any class that inherits from α . In other words, any object in the deep extent of α can be used where a direct instance of a class α is expected.

In the example, we can state the following class conformity relation: (1) X conforms to X, (2) Y conforms to Y and X, (3) Z conforms to Z, Y and X, (4) W conforms to W and X. Thus, a direct instance of X can be substituted by a direct instance of Y, Z or W, while a direct instance of Y can be substituted by a direct instance of Z.

Moreover, we have that the conformity relation is transitive but not necessarily antisymmetric, i.e., given two classes α and β , α conforms to β and β conforms to α does not imply α and β are the same class. For example, it is possible that W conforms to Y and that Y conforms to W.

Is-a Relation

Given two classes α and β , we say that α “is a” β , and denote it as $\alpha \leq_\tau \beta$, either if α and β are the same class or if α inherits from β . Thus, $\alpha \leq_\tau \beta$ implies α conforms to β and, therefore, $\alpha \leq_\tau \beta$ implies a direct instance of α can be used where a direct instance of β is expected. For example, $Y \leq_\tau X$ and, in fact, a direct instance of Y can substitute a direct instance of X .



$$\alpha \leq_\tau \beta \Rightarrow \alpha \text{ conforms to } \beta$$

The is-a relation is reflexive and, since the inheritance relation is transitive, the is-a relation is also transitive. For example, we can state that: (1) $X \leq_\tau X$, (2) $Y \leq_\tau Y$, (3) $Y \leq_\tau X$, (4) $Z \leq_\tau Z$, (5) $Z \leq_\tau Y$, (6) $Z \leq_\tau X$, (7) $W \leq_\tau W$, (8) $W \leq_\tau X$.

In addition, because a class hierarchy is an acyclic graph (directed tree), the is-a relation is antisymmetric. For example, if class $\alpha \leq_\tau X$ and $X \leq_\tau \alpha$ then α is X . Therefore, we have that the is-a relation is reflexive, antisymmetric and transitive, which means that a set of classes is *partially ordered* with respect to the is-a relation.

Generalisation/Specialisation

Since a class α that inherits from a class β also conforms to β and possibly adds some specifications, we say that α is a *specialisation* of β and, conversely, β is a *generalisation* of α . Thus, the inheritance relation is also referred to as a generalisation/specialisation relation, and a class hierarchy is also referred to as

a generalisation/specialisation hierarchy.

Moreover, because classes are types, α is a subtype of β . Therefore, we say that α is a *subclass* of β and, conversely, β is a *superclass* of α . Accordingly, we differentiate between direct and indirect subclasses and between direct and indirect superclasses. For example, Y is direct subclass of X while Z is indirect subclass of X and, conversely, X is direct superclass of Y while X is indirect superclass of Z. Finally, for an object that is a direct instance of a class α , we say that α is the *most specific class* of the object.

4.6 Summary

The schema for bibliographical references shown in Figure 4.13 illustrates the main concepts of the object model.

Association

There is an association between **Individual** and **Article**, meaning:

- The role of **Individual** in the association is **Author**.
 - The role of **Article** in the association is **Article** since there is no other specification.
 - The multiplicity of **Individual** is $[1,n]$. So, an instance of **Article** is associated to one or many instances of **Individual**, i.e., an article has one or many authors.
-

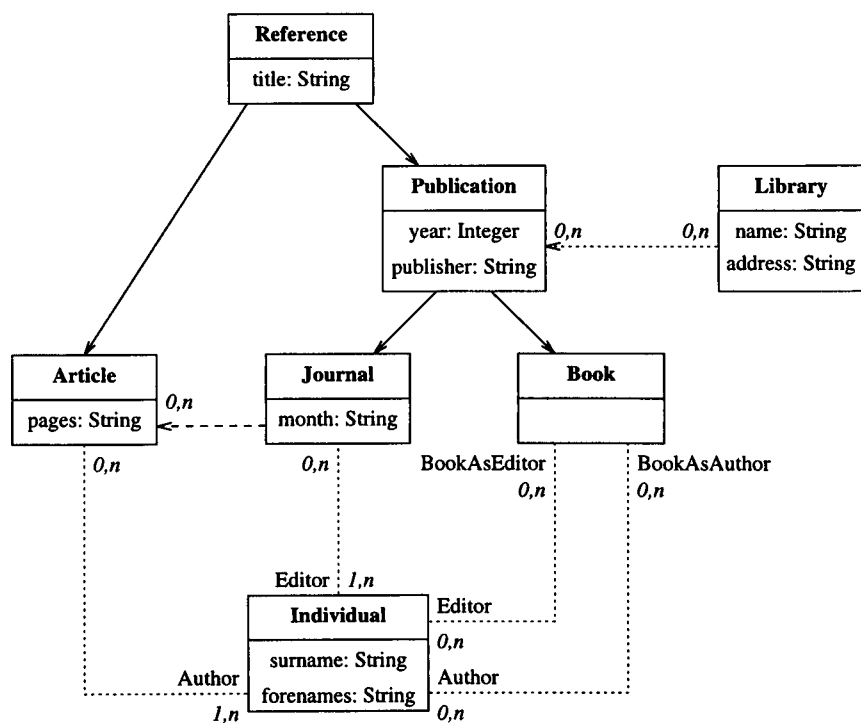


Figure 4.13 Simple schema for bibliographical references

- The multiplicity of **Article** is $[0,n]$. So, an instance of **Individual** is associated to none or many instances of **Article**, i.e., an individual is author of any number of articles.

Tight Aggregation

There is a tight aggregation between **Journal** and **Article**, meaning:

- An instance of **Article** is physically part of an instance of **Journal**.
- An instance of **Journal** is an aggregate of any number of instances of **Article**.

Loose Aggregation

There is a loose aggregation between **Library** and **Publication**, meaning:

- An object that is an instance of **Publication** is conceptually part of any number of instances of **Library**.
- An instance of **Library** is an aggregate of any number of instances of **Publication**.

Class Hierarchy

There is a class hierarchy defined by the classes **Reference**, **Publication**, **Article**, **Journal** and **Book**, meaning:

- **Reference** is superclass of **Article** and **Publication**. **Reference** defines the attribute **title** which is common to **Article** and **Publication**.
 - **Article** defines the attribute **pages**. So, an instance of **Article** has the attribute **title** inherited from **Reference** and the attribute **pages**.
 - **Publication** is superclass of **Journal** and **Book**. **Publication** defines the attributes **year** and **publisher** which are common to **Journal** and **Book**. So, an object that is a instance of **Publication** has the attribute **title** inherited from **Reference**, and the attributes **year** and **publisher**.
 - **Journal** defines the attribute **month**. So, an instance of **Journal** has the attributes **title**, **year** and **publisher** inherited from **Publication**, and the specific attribute **month**.
-

- **Book** defines no attribute. So, a direct instance of **Book** has the same attributes as direct instances of its direct superclass, i.e., the attributes **title**, **year** and **publisher** inherited from **Publication**.
- **Journal** and **Book** inherit (from **Publication**) the loose aggregation with **Library**. So, instances of **Journal** and **Book** are conceptually part of instances of **Library**.
- **Article**, **Publication**, **Journal** and **Book** are subclasses of **Reference**. So, a direct instance of any of these classes can substitute a direct instance of **Reference**.

4.7 Conclusions

The technique for object modelling described in this Chapter makes use of a simple set of concepts which are becoming common amongst object-oriented systems. Despite its simplicity, the technique permits modelling of information objects with considerably great expressiveness and re-use of definitions by applying inheritance, as the examples have shown. An advantage of being simple is that a system that supports the technique is perfectly feasible by employing standard programming languages and operating systems, thereby making it possible to inter-operate with other object-oriented systems. The same approach is taken by the OMG CORBA [45], for example. Another clear advantage is that the technique can be easily assimilated and applied.

The graphic notation defined permits a concise and unambiguous representation of object properties. The only aspect of object-oriented modelling which cannot be represented through the graphic notation is method semantics, or the

behaviour of the objects. We believe that a distinct form of notation should be used for specifying method semantics as it is equivalent to specifying an algorithm. For example, a notation for formal specification of software, such as the Z Notation [53], could be employed as a complement to the graphic notation.

Finally, we have emphasised the importance of attributing semantics to pointers between objects and have introduced *loose aggregation* in addition to the set of concepts normally found in object-oriented modelling techniques. This feature of our technique, in particular, is going to reveal very important in Chapter 6 where the technique is employed to represent its own elements (classes, attributes, methods and relationships) and a model to organise the class space which is devised in Chapter 5 (schemas).

CHAPTER 5

Class Space Organisation

In this Chapter we introduce a means of organising the class space by which classes are grouped in *schemas* that can be composed of sub-schemas, recursively. Since a class designates a set of objects (the deep extent of the class), a schema indirectly permits the selection of a set of objects for manipulation, which we define in this Chapter as *databases*. Moreover, classes define object properties which can be inherited by subclasses, thereby permitting the re-use of type definitions. Thus, schemas permit the organisation of the class space for both the organisation of objects and the management of types. In general, the motivation for having schemas as a means of organising classes include the following items:

1. *Security*: A schema can be used for defining the set of objects (at the class granularity) that each user should be able to get access.
 2. *Customisation*: A schema can be used for selecting only the classes which are of interest to users.
 3. *Efficiency*: A schema can be used for selecting a specific set of classes for manipulation, thereby reducing the type information necessary to be loaded by programs.
-

4. *Administration*: Schemas document which classes are defined thereby permitting to manage their use and re-use.
5. *Scalability*: Proper class organisation is particularly important when the number of classes, objects and users are considerably large, distributed and require decentralised administration. Schemas permit the partition of the class space in a hierarchical structure, which is the approach normally taken in scalable systems [36]. In Chapter 7 we define *views* as a means of associating schema information and index information with corresponding objects in order to organise the object space.

We informally introduce the notions of schema and database in our context and then give formal definitions for them. The formal definitions are presented for the following reasons:

1. The formal models for databases found in the literature define schemas as global entities, i.e., the schema space is normally flat. Since we organise schemas in hierarchies (for scalability purposes) we formalise this new concept. Moreover, the formal definition of schema is necessary in Chapter 6 for defining the *meta-schema*.
 2. The formal models for databases found in the literature define databases as a “consistent” set of objects, i.e., a set of objects where all object references are to objects that also belong to the set (referential integrity or no dangling identifier assumption [31]). However, since they do not support the notion of sub-schemas, they do not consider the case where a set of objects is “relatively consistent”, i.e., the case where a set is consistent with respect to the relationships included by a sub-schema, rather than “absolutely consistent”.
-

We introduce a new concept, namely *relative self-containment*, to differentiate between absolute and relative referential integrity and then formally define database. The main advantage of having relative self-containment is to permit a set of objects (a database) to be consistently manipulated even if they have (*invisible*) dangling references.

5.1 Overview

A schema is a collection of classes, including all corresponding inheritance arcs and relationships. For example, the diagram in Figure 4.13 represents a simple schema for bibliographical references. The only invariant that must hold in a schema is that, recursively, the superclasses of every class in the schema must be in the schema, too.

For this reason, we define **root-subtree** as a subtree of a class hierarchy such that the root of the subtree is the root of the class hierarchy. In other words, the root of a root-subtree has no superclass. For example, in the schema for bibliographical references, a subtree that has either **Reference**, **Library** or **Individual** as its root is a root-subtree, otherwise it is not.

Thus, obviously, the empty set defines a schema and, consequently, given a set of class hierarchies, a schema can be any union of root-subtrees of the class hierarchies. Furthermore, any union of schemas is a schema. For example, any union of the following sets of classes defines a schema.

$s_0 = \emptyset$	$s_4 = \{\text{Reference, Publication, Journal}\}$
$s_1 = \{\text{Reference}\}$	$s_5 = \{\text{Reference, Publication, Book}\}$
$s_2 = \{\text{Reference, Article}\}$	$s_6 = \{\text{Library}\}$
$s_3 = \{\text{Reference, Publication}\}$	$s_7 = \{\text{Individual}\}$

Schema Aggregation

Schemas can be recursively aggregated to form larger schemas. For this reason, a schema is recurrently defined as an aggregation containing a root-subtree that can be nil and a set of schemas. Thus, directly, a schema can contain at most one root-subtree and, indirectly, it can contain any number of root-subtrees. The schema for bibliographical references, for example, contains three class hierarchies defined by the root classes **Reference**, **Library** and **Individual** and, consequently, the schema is necessarily composed of smaller schemas.

A schema that contains only a root-subtree is referred to as a **basic schema**. Moreover, aggregate and component schemas are, respectively, referred to as **super-schema** and **sub-schema**. Thus, given a schema s , if its set of schemas is empty then s is a basic schema, otherwise s is a super-schema and each schema in the set of schemas of s is a sub-schema of s . Finally, two schemas are *equivalent* if they contain the same set of classes, independently of their internal arrangement of sub-schemas.

As an example, Table 5.1 shows some possible definitions of schemas and their aggregations using the classes in the schema for bibliographical references, which is

schema	root-subtree (set of classes)	sub-schemas
w_1	{ Reference, Article, Publication, Journal, Book }	\emptyset
w_2	{ Library }	\emptyset
w_3	{ Individual }	\emptyset
w_4	\emptyset	{ w_1, w_2, w_3 }
w_5	{ Reference, Article, Publication, Journal, Book }	{ w_2, w_3 }
w_6	{ Library }	{ w_1, w_3 }
w_7	{ Individual }	{ w_1, w_2 }

Table 5.1 Example of schema aggregation

necessarily a super-schema. We can note that: (1) w_1 , w_2 and w_3 are basic schemas since they do not contain sub-schemas, (2) w_4 is a super-schema containing three sub-schemas, (3) w_5 , w_6 and w_7 are super-schemas containing a root-subtree and two sub-schemas, (4) w_4 , w_5 , w_6 and w_7 are equivalent because they contain the same set of classes.

Self-contained Schema

Schemas are useful for defining portions of a database for manipulation because every schema has implicitly associated with it a set of objects: the union of the deep extents of all the classes that belong to the schema. Although, in principle, schemas can be freely defined and aggregated, certain constraints are necessary to ensure that the set of objects associated with a certain schema is self-contained.

A schema does not necessarily ensure referential integrity. It is possible to define a schema so that the associated set of objects contains objects with dangling

references, i.e., references to objects which are not in the set. In other words, the set of objects associated to a schema is not necessarily a database. For example, the set of objects associated to the schema w_2 in Table 5.1, consisting only of instances of the class **Library**, is not a database since an instance of **Library** can contain references to instances of the class **Publication**, which do not belong to the set of objects defined by w_2 .

Therefore, we define **self-contained schema** as a schema s such that for every class α that belongs to s all superclasses and all classes related to α belong to s , recursively. This constraint ensures that the set of objects associated to a self-contained schema does not contain dangling references. Some examples of self-contained schemas are the schemas w_4 , w_5 , w_6 and w_7 in Table 5.1. Another example is a super-schema containing the classes **Library**, **Publication** and **Reference**.

5.2 Schema Definition

Graph Representation

Classes and corresponding inheritance arcs can be represented by a directed graph; every vertex of the graph represents a class and every arc (directed edge) of the graph represents an inheritance arc. (In Appendix A we prove that such a graph is a directed tree.) We define a notation for graphs which is summarised in Figure 5.1 through an example.

Notation 5.1 A directed graph G is denoted by a doublet (V, A) , where V is the set of vertices and A is the set of arcs of G . □

Notation 5.2 Given a directed tree Ψ , the notation $Root(\Psi)$ denotes the vertex in $\Psi.V$ which is the root of Ψ . **Root**

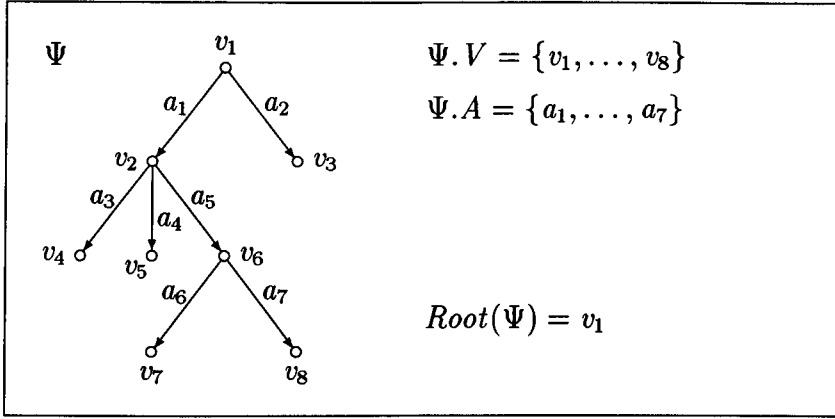


Figure 5.1 Graph notation

Also, we define notation to denote all classes, all inheritance arcs and then the graph that represents all classes and inheritance arcs.

Notation 5.3 The symbol \mathcal{C} denotes the set of all classes. **C**

Notation 5.4 The symbol \mathcal{A} denotes the set of all inheritance arcs between classes in \mathcal{C} . **A**

Notation 5.5 The symbol \mathcal{G} denotes a directed graph such that $\mathcal{G}.V = \mathcal{C}$ and $\mathcal{G}.A = \mathcal{A}$. **G**

Self-contained Set of Classes

A schema corresponds to a subgraph of \mathcal{G} that can be used independently. However, a subgraph of \mathcal{G} corresponds to a set of classes, which is not necessarily

independent of the remaining classes. A set of classes C is *self-contained*, hence independent of the remaining classes in \mathcal{C} , if all classes referred to by the classes in C also belong to C . According to our definitions of class and relationship,¹ a class x contains a reference to a class y in one of the following situations:

1. x is direct subclass of y

In this case, x contains the name of y as the name of its superclass.

2. x and y are related classes

In this case, x contains the name of y as the name of a related class.

Thus, more specifically, a set of classes C is *totally* self-contained if, and only if, the following conditions hold:

S_1 . The superclass of every class in C belongs to C .

S_2 . Every class related to every class in C belongs to C .

If S_1 holds we say that C is *self-contained with respect to hierarchy*. If S_2 holds we say that C is *self-contained with respect to relationship*.

Definition 5.1 (Class Hierarchy Self-containment) A set of classes C is *self-contained with respect to hierarchy* iff $\forall x \in C$: if x is derived from a class y then $y \in C$.

□

Notation 5.6 Given a class x , the notation $\rho(x)$ denotes the set of all classes related to x .

¹Definition A.23 (Class) and Definition A.8 (Relationship Specification) in Appendix A.

Definition 5.2 (Class Relationship Self-containment) A set of classes C is self-contained with respect to relationship iff $\forall x \in C : \rho(x) \subseteq C$. \square

Definition 5.3 (Totally Self-contained Set of Classes) A set of classes C is totally self-contained iff:

- C is self-contained with respect to hierarchy
- C is self-contained with respect to relationship \square

Example 5.1 Let us consider the classes represented by the graph depicted in Figure 5.2, where class relationships are represented by dashed lines, and references between classes are listed on the right-hand side of the graph: the first and the second columns, respectively, contain the superclass and the set of related classes for each class in the graph. Each row of Table 5.2 contains a subset of the classes in the Figure and an indication (\checkmark) whether the set is self-contained with respect to hierarchy and relationship. \diamond

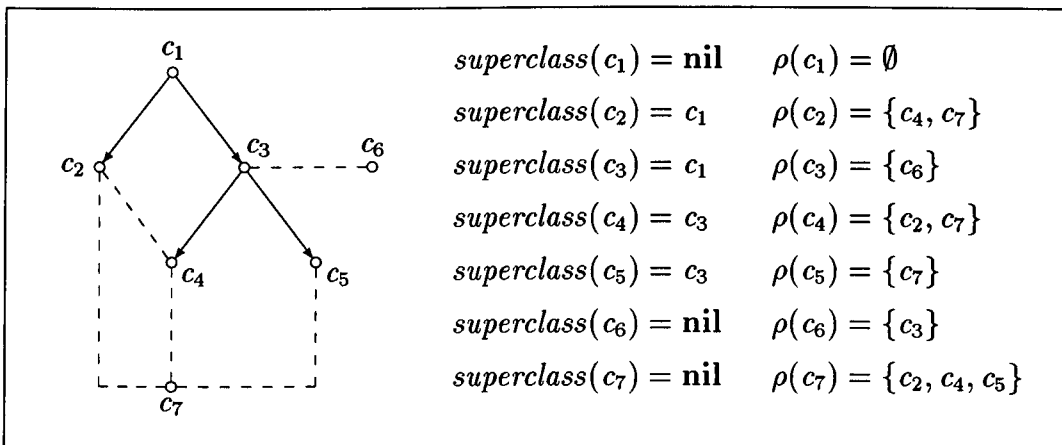


Figure 5.2 Example of references between classes

set of classes	self-containment	
	hierarchy	relationship
$\{c_1\}$	✓	✓
$\{c_1, c_2, c_3\}$	✓	
$\{c_3, c_6\}$		✓
$\{c_3, c_4, c_5\}$		
$\{c_1, c_3, c_6\}$	✓	✓
$\{c_1, \dots, c_7\}$	✓	✓

Table 5.2 Example of self-contained set of classes

Root-subtree

Since the (connected) components of \mathcal{G} are (tree-structured) class hierarchies, the components of a subgraph of \mathcal{G} are subtrees of class hierarchies. According to condition S_1 , in a subgraph that designates a schema, the set of classes in such a subtree must be self-contained with respect to hierarchy. For simplicity, we designate such a subtree as self-contained.

Definition 5.4 (Self-contained Subtree) *A subtree H of a class hierarchy Ψ in \mathcal{G} is self-contained iff the set of classes $H.V$ is self-contained with respect to hierarchy.*

□

We now define *root-subtree* as a subtree of a class hierarchy such that the root of the subtree is the root of the class hierarchy (i.e., the class which is the root of a root-subtree has no superclass). A root-subtree is self-contained and, conversely, that a self-contained subtree is a root-subtree, as we state in Theorem 5.1 and

prove in Appendix B.

Definition 5.5 (Root-subtree) A root-subtree H is a subtree of a class hierarchy Ψ in \mathcal{G} such that $\text{Root}(H) = \text{Root}(\Psi)$. \square

Example 5.2 Let us consider the classes represented by the graph depicted in Figure 5.3. A root-subtree, such as H_1 , is denoted by the dotted-line rectangle and a black point connected to the rectangle by another dotted line; the rectangle surrounds the vertices and arcs of the root-subtree. Since a root-subtree is a graph, the classes of a root-subtree is given by its set of vertices, such as $H_1.V$ for the root-subtree H_1 . The list on the right-hand side of the graph shows the set of classes of H_1 and some other possible root-subtrees, namely H_2, \dots, H_5 , which, for simplicity, are not denoted in the graph. We should notice that any root-subtree must contain the class c_1 since it is the only root class in the graph. \diamond

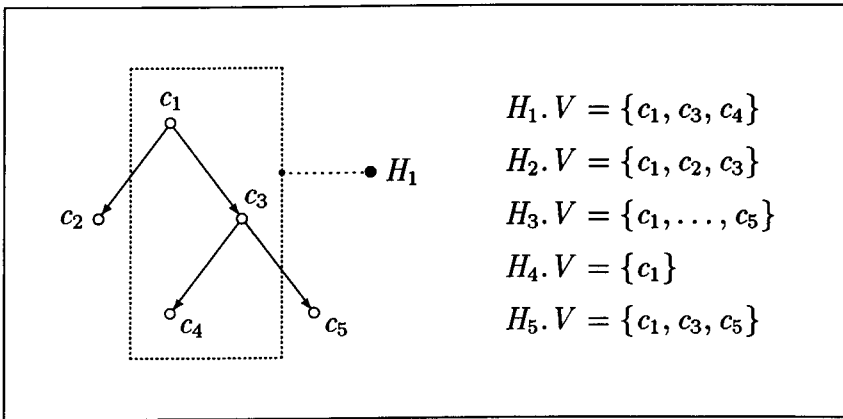


Figure 5.3 Example of root-subtree

Theorem 5.1 (Root-subtree Self-containment) A subtree H of a class hierarchy Ψ in \mathcal{G} is self-contained iff H is a root-subtree. \blacksquare

Schema

A schema is recurrently defined as an aggregation containing a root-subtree which can be nil and a set of schemas; directly, a schema may contain at most one root-subtree and, indirectly, it may contain any number of root-subtrees. Thus, a schema corresponds to a set of classes that is self-contained with respect to hierarchy but not necessarily self-contained with respect to relationship. Let us assume the existence of a countably infinite set WN of schema names, then we can define schema as follows.

WN

Definition 5.6 (Schema) *A schema is a triple (n, H, S) , where:*

- $n \in WN$
- H is either a root-subtree or **nil**
- S is a set of schemas

Terminology:

- Let w be a schema. If $w.H \neq \mathbf{nil}$ and $w.S = \emptyset$ then w is a basic schema.
- Let w_1 and w_2 be schemas. If $w_2 \in w_1.S$ then w_1 is super-schema of w_2 , while w_2 is sub-schema of w_1 . □

Example 5.3 Let us consider the classes represented by the graph depicted in Figure 5.4. A schema, such as w_5 (bottom of Figure), is denoted by a full-line rectangle and arrows connecting the rectangle to a root-subtree, such as H_5 , and other schemas, such as w_4 . Root-subtrees are surrounded by dotted-lines, and class relationships are denoted by dashed lines. Let us suppose that s_1, \dots, s_6 are schema names, then we have the following elements in the graph:

- Class hierarchies Ψ_1, Ψ_2, Ψ_3 :

$$\begin{array}{ll} \Psi_1.V = \{c_1, \dots, c_8\} & \text{Root}(\Psi_1) = c_1 \\ \Psi_2.V = \{c_9, \dots, c_{11}\} & \text{Root}(\Psi_2) = c_9 \\ \Psi_3.V = \{c_{12}, \dots, c_{15}\} & \text{Root}(\Psi_3) = c_{12} \end{array}$$

- Root-subtrees H_1, \dots, H_5 :

$$\begin{array}{ll} H_1.V = \{c_1, \dots, c_8\} & H_4.V = \{c_{12}, c_{14}\} \\ H_2.V = \{c_9, \dots, c_{11}\} & H_5.V = \{c_1, c_2, c_5, c_6\} \\ H_3.V = \{c_{12}, \dots, c_{15}\} & \end{array}$$

- Basic schemas w_1, \dots, w_4 , and super-schemas w_5, w_6 :

$$\begin{array}{ll} w_1 = (s_1, H_1, \emptyset) & w_4 = (s_4, H_4, \emptyset) \\ w_2 = (s_2, H_2, \emptyset) & w_5 = (s_5, H_5, \{w_4\}) \\ w_3 = (s_3, H_3, \emptyset) & w_6 = (s_6, \text{nil}, \{w_1, w_2, w_3\}) \end{array}$$

◇

Schema Name Distinction

The name of a schema must be distinct from the name of any other schema to permit each one to be referred to unambiguously. We define a notation to denote the set of all schemas and formally define that each schema has a distinct name.

Notation 5.7 The symbol \mathcal{W} denotes the set of all schemas. □



Invariant 5.1 (Schema Name Distinction) $\forall x, y \in \mathcal{W}$: if $x.n = y.n$ then $x = y$. ♦

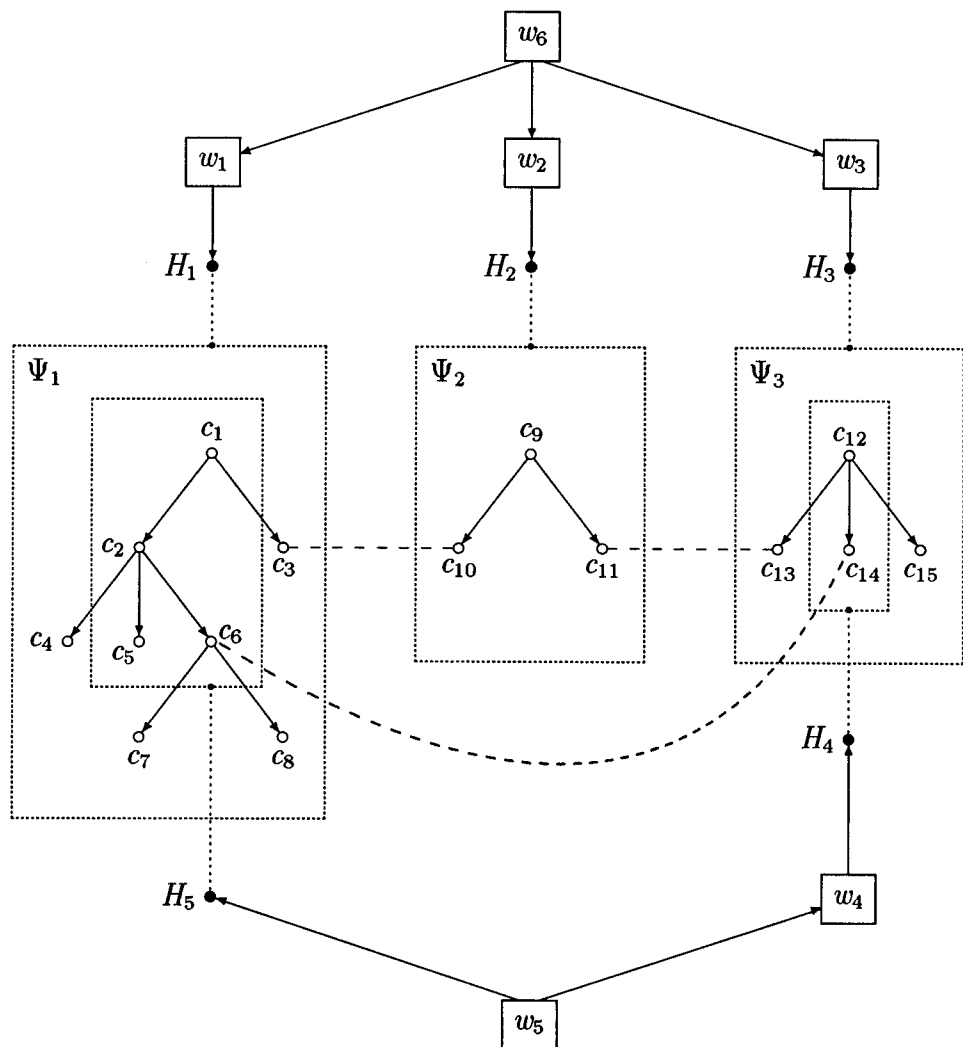


Figure 5.4 Example of schema aggregation

Acyclic Arrangement of Schemas

According to Definition 5.6 (Schema), a schema may contain a number of schemas, recursively, hence forming a hierarchy of aggregated schemas. Obviously, a schema cannot contain itself (neither directly nor indirectly), otherwise the recursion is infinite. In other words, there can be no cycle in a hierarchy of schemas. We define a notation to denote all sub-schemas of a schema and formally define that a schema cannot be sub-schema of itself, i.e., the arrangement of schemas is acyclic.

Notation 5.8 Given a schema w , the notation $\psi(w)$ denotes the set of all sub-schemas of w , recursively: □

$$\psi(w) = w.S \cup \bigcup_{x \in w.S} \psi(x) \quad \square$$

Invariant 5.2 (Acyclic Arrangement of Schemas) $\forall w \in \mathcal{W} : w \notin \psi(w)$. ◆

Self-contained Schema

The set of classes corresponding to a schema is self-contained with respect to hierarchy since a schema is composed of root-subtrees. In addition, such a set of classes may also be self-contained with respect to relationship, hence totally self-contained. For simplicity, we say that a schema is self-contained if its corresponding set of classes is totally self-contained, otherwise we say that the schema is not self-contained. We define a notation to denote the set of classes that corresponds to a schema and formalise self-contained schema.

Notation 5.9 Given a schema w , the notation $\Phi(w)$ denotes the set of classes in □

w , recursively:

$$\Phi(w) = w.H.V \cup \bigcup_{x \in w.S} \Phi(x) \quad \square$$

Definition 5.7 (Self-contained Schema) A schema w is self-contained iff the set of classes given by $\Phi(w)$ is totally self-contained. \square

Example 5.4 Let us consider again the schemas depicted in Figure 5.4. The set of classes corresponding to each schema and the sets of related classes which are not empty are given as follows.

$$\begin{aligned} \Phi(w_1) &= H_1.V = \{c_1, \dots, c_8\} \\ \Phi(w_2) &= H_2.V = \{c_9, c_{10}, c_{11}\} \\ \Phi(w_3) &= H_3.V = \{c_{12}, \dots, c_{15}\} \\ \Phi(w_4) &= H_4.V = \{c_{12}, c_{14}\} \\ \Phi(w_5) &= H_5.V \cup \Phi(w_4) = \{c_1, c_2, c_5, c_6, c_{12}, c_{14}\} \\ \Phi(w_6) &= \Phi(w_1) \cup \Phi(w_2) \cup \Phi(w_3) = \{c_1, \dots, c_{15}\} \end{aligned}$$

$$\begin{aligned} \rho(c_3) &= \{c_{10}\} & \rho(c_6) &= \{c_{14}\} & \rho(c_{10}) &= \{c_3\} \\ \rho(c_{11}) &= \{c_{13}\} & \rho(c_{13}) &= \{c_{11}\} & \rho(c_{14}) &= \{c_6\} \end{aligned}$$

Therefore, self-containment of the given schemas is given as follows.

$$\begin{aligned} c_3 \in \Phi(w_1) \wedge \rho(c_3) \not\subseteq \Phi(w_1) &\Rightarrow w_1 \text{ is not self-contained} \\ c_8 \in \Phi(w_1) \wedge \rho(c_8) \not\subseteq \Phi(w_1) &\Rightarrow w_1 \text{ is not self-contained} \\ c_{10} \in \Phi(w_2) \wedge \rho(c_{10}) \not\subseteq \Phi(w_2) &\Rightarrow w_2 \text{ is not self-contained} \\ c_{11} \in \Phi(w_2) \wedge \rho(c_{11}) \not\subseteq \Phi(w_2) &\Rightarrow w_2 \text{ is not self-contained} \end{aligned}$$

$c_{13} \in \Phi(w_3) \wedge \rho(c_{13}) \not\subseteq \Phi(w_3) \Rightarrow w_3$ is not self-contained

$c_{14} \in \Phi(w_3) \wedge \rho(c_{14}) \not\subseteq \Phi(w_3) \Rightarrow w_3$ is not self-contained

$c_{14} \in \Phi(w_4) \wedge \rho(c_{14}) \not\subseteq \Phi(w_4) \Rightarrow w_4$ is not self-contained

$\Phi(w_5)$ is totally self-contained $\Rightarrow w_5$ is self-contained

$\Phi(w_6)$ is totally self-contained $\Rightarrow w_6$ is self-contained

5.3 Database Definition

A schema designates a set of classes and each class designates a set of objects: the deep extent of the class. Thus, a schema (indirectly) designates a set of objects: the union of all deep extents of the classes designated by the schema. Firstly we introduce a notation to designate the deep extent of a class and then we introduce a notation to denote the set of objects designated by a set of classes.

Notation 5.10 Given a class c , the notation $Ext^*(c)$ denotes the deep extent of c . Ext^*

□

Notation 5.11 Given a set of classes C , the notation $\Omega(C)$ denotes the set of all objects which are instances of classes in C : Ω

$$\Omega(C) = \bigcup_{c \in C} Ext^*(c)$$

□

A set of objects S where all objects referred to by objects in S also belong to S , i.e., there are no “dangling” references in S , can be manipulated independently of the remaining objects. Accordingly, we define such a set of objects as *absolutely*

self-contained. Firstly we define a notation to denote the set of all objects and another notation to denote the set of objects related to a given object.

\mathcal{O}

Notation 5.12 *The symbol \mathcal{O} denotes the set of all objects.* □

Ref

Notation 5.13 (Set of Related Objects) *Given an object $o \in \mathcal{O}$, the notation $Ref(o)$ denotes the set of all objects which are related to o .* □

Definition 5.8 (Absolute Self-containment) *A set of objects S is absolutely self-contained iff $\forall o \in S : Ref(o) \subseteq S$.* □

However, a set of objects does not need to be absolutely self-contained to be independent for manipulation purposes. For example, let us consider the classes and respective instances depicted in Figure 5.5. For this discussion we labelled the objects as 1, 2 and 3, and annotated their parts corresponding to each class. Thus, the object 1 has a part C, the object 2 has parts A and B, and the object 3 has a part D. The object relationship r corresponds to the class relationship R , while the object relationship s corresponds to the class relationship S . Since R is between C and B, r is maintained by the parts C and B of the objects 1 and 2, respectively. Similarly, since S is between A and D, s is maintained by the parts A and D of the objects 2 and 3. Now, let us suppose that a self-contained schema w is defined containing the classes A and D. The set of objects designated by w includes the objects 2 and 3. Since w does not include class B, the part B of object 2 is not “covered” by w , i.e., if a user manipulates the object 2 through w then the part B of object 2 is not “seen” by the user. Consequently, the object relationship r is not relevant to the user. Thus, although the set of objects designated by w is obviously not absolutely self-contained, this set contains no dangling references

with respect to w . In other words, the set of objects designated by w is self-contained *relatively* to the class relationships included by w . The set of objects designated by a self-contained schema, in particular, is referred to as *database*. Thus, a database is self-contained with respect to its designating schema.

Definition 5.9 (Database) Given a self-contained schema w , the database with respect to w , denoted as $DB(w)$, is the set of objects which are instances of the set of classes in w : **DB**

$$DB(w) = \Omega(\Phi(w)) \quad \square$$

Definition 5.10 (Relative Self-Containment) Given a self-contained schema w , a set of objects S is self-contained with respect to w if $S \subseteq DB(w)$. \square

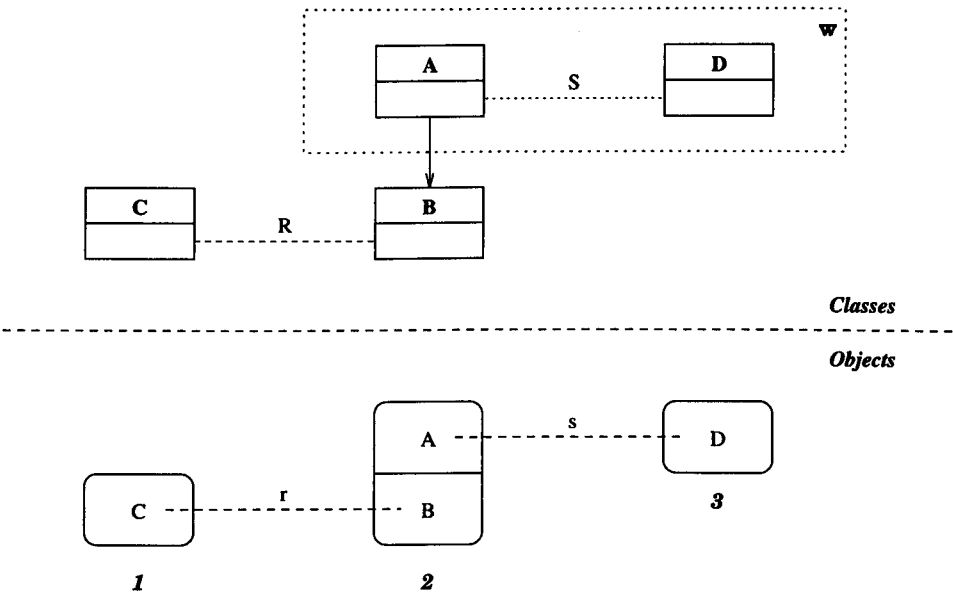


Figure 5.5 Relative self-containment of schemas

5.4 Conclusions

The recursive definition of schema and their consequent hierarchical arrangement provide a powerful means of organising classes and objects: schemas can be freely decomposed in sub-schemas and, conversely, they can be freely aggregated to other schemas to form larger schemas. This approach permits flexible and scalable organisation of class and object space, as we discuss in Chapter 7, in contrast with the flat space normally offered by database systems.

In addition, with the new concept that we have introduced, *relative self-containment*, a schema can designate a set of objects — a database — for manipulation independently of the remaining objects even if an object in the database contains a reference to an object which does not belong to the database. The only requirement is that the references that correspond to the schema are self-contained. This approach relaxes the traditional referential integrity required in object-oriented database systems.

CHAPTER 6

Meta-object Model

Schemas (including classes, attributes, methods and relationships) themselves constitutes information which require proper management. In this Chapter, we show that the object model is *reflexive* by defining a set of special (reserved) classes — the *meta-classes* — which can represent schemas. Next we define a (self-contained) schema — the *meta-schema* — that contains all meta-classes. Thus, the database designated by the meta-schema — the *meta-objects* — represents all (meta) information about both the predefined and user-defined schemas. We demonstrate the correctness of the meta-schema through a set of rules for mapping schema elements to meta-objects. Since meta-objects permit representation and manipulation of schemas, we refer to the collection of definitions introduced in this Chapter as *meta-object model*.

A database of meta-objects corresponds to the notion of *data dictionary* or *meta-data* often employed in database and CASE systems; meta-objects provide for system administration, documentation, and software management. Systems administrators use meta-objects to manage schemas: creation, modification and deletion of classes, and their organisation in schemas and sub-schemas. As a source of documentation meta-objects permit users to learn what classes exist,

thereby making it easier for them to formulate queries and discover information. The information about types¹ maintained by meta-objects permits automatic generation of program code for several purposes. The code corresponding to classes, which is necessary to manipulate objects from programs, can be generated and then linked to specific applications, such as information browsers, query interpreters and report generators. Also, the type information in meta-objects can be used to generate specific code for object state checkpointing² and transport within network messages in distributed systems. We discuss the use of meta-objects for code generation in Chapter 8 where an implementation of our model is described.

In addition, we make a non-conventional use of meta-objects: query resolution. The information about classes (basically class name, attributes, inheritance and relationships) maintained by meta-objects naturally permits appropriate type checking of query expressions (e.g. verify if an attribute which is specified in a query expression as belonging to a certain class really belongs to it). Moreover, we add to the normal information represented by meta-objects the information about indices which is necessary for resolving queries. Thus, every object operation that affects index information is followed by an operation that uses meta-objects to locate the indices that should be updated, and a query interpreter solves queries by simply traversing meta-objects and obtaining information from the corresponding indices. The use of meta-objects for query resolution purposes is explained in Chapter 8.

¹In our model a class is a type.

²Typically, transaction-based systems store object states in auxiliary storage for recovery purposes.

Meta-objects also have a fundamental role in the formal definition of our model: schema space definition. Conceptually, schema elements exist only if the meta-objects that map them exist. For example, every class has to be mapped to an appropriate set of meta-objects in order to register it as a valid type, thereby enabling the creation of instances of it. In this respect, our meta-object model differs from “pure” object-oriented systems, such as Smalltalk[23] and its variations, where a class is an object. In our model a class is simply a type which is *represented by* meta-objects as a means of implementing the type space. This poses an interesting problem of solving the implicit recursion in the meta-model. Because every class has to be mapped to meta-objects for it to exist, it requires the existence of meta-classes in order to enable the creation of meta-objects. However, meta-classes are classes which also need to be mapped for them to exist. In this Chapter, we simply postulate the existence of the meta-classes and, then, in Chapter 8 we explain how this recursion problem is solved.

Notation

The meta-object model has its basis on the object model introduced in Chapter 4 and formally defined in Appendix A. Because that formal definition is considerably detailed for the purpose of explaining the meta-model, here we repeat only the essential notation introduced and simplify its format for readability reasons. However, in Appendix D we demonstrate that the meta-object model complies with the formalism defined for the object-model by giving a version that makes use of the formal notation for each definition and example presented in this Chapter.

- $class(n)$: the class whose name is n
- $className(c)$: the name of the class c
- $objClassName(o)$: the name of the most specific class of the object o
- $schemaName(w)$: the name of the schema w
- $Att(c)$: the set of all attributes of the class c
- $Rel(c)$: the set of all relationships of the class c
- $Met(c)$: the set of all methods of the class c
- $o \dashrightarrow n$: the value of the attribute n of the object o
- $x \overset{Y}{\underset{X}{\rightleftarrows}} y$: the relationship between the objects x and y where the role of x is X and the role of y is Y
- $\xi(n)$: the deep extent of the class whose name is n
- $relType(r)$: the type of the relationship r (“Loose Aggregation”, “Tight Aggregation” or “Association”)

6.1 Meta-schema

The meta-schema is depicted in Figure 6.1. The basic principle is to represent schemas (and their components) as inter-related instances of the meta-classes. For example, if there is a user-defined class **Person** then there must be an instance, say o_1 , of the meta-class **Class** whose attribute **name** has value **Person**.

$$className(o_1) = \text{Class} \quad o_1 \dashrightarrow \text{name} = \text{Person}$$

If the class **Person** has an attribute **age** of type **Integer** then there must be an instance, say o_2 , of the meta-class **IntegerAttribute** whose attribute name is **age**.

$$className(o_2) = \text{IntegerAttribute} \quad o_2 \dashrightarrow \text{name} = \text{age}$$

Moreover, o_2 must be related to o_1 in order to represent that the attribute o_2 is a component of the class o_1 .

$$o_1 \overset{\text{Attribute}}{\underset{\text{Class}}{\rightleftarrows}} o_2$$

Self Representation

The basic principle of representing user-defined schemas as instances of meta-classes can be applied to the meta-schema as well, i.e., we can have a database of meta-objects that represents the meta-schema. For example, since there is a class **Class** there must be an instance, say o_3 , of the class **Class** whose attribute name has value **Class**. Also, since the class **Class** has an attribute **name** of type **String** there must be an instance, say o_4 , of the meta-class **StringAttribute** whose attribute **name** has value **name**. Moreover, attribute o_4 must be related to class o_3 .

$$className(o_3) = \text{Class} \quad o_3 \dashrightarrow \text{name} = \text{Class}$$

$$className(o_4) = \text{StringAttribute} \quad o_4 \dashrightarrow \text{name} = \text{name}$$

$$o_3 \overset{\text{Attribute}}{\underset{\text{Class}}{\rightleftarrows}} o_4$$

As another example, there must be two instances, say o_5 and o_6 , of the meta-class **Class** whose attributes **name** have values **Attribute** and **StringAttribute**, respec-

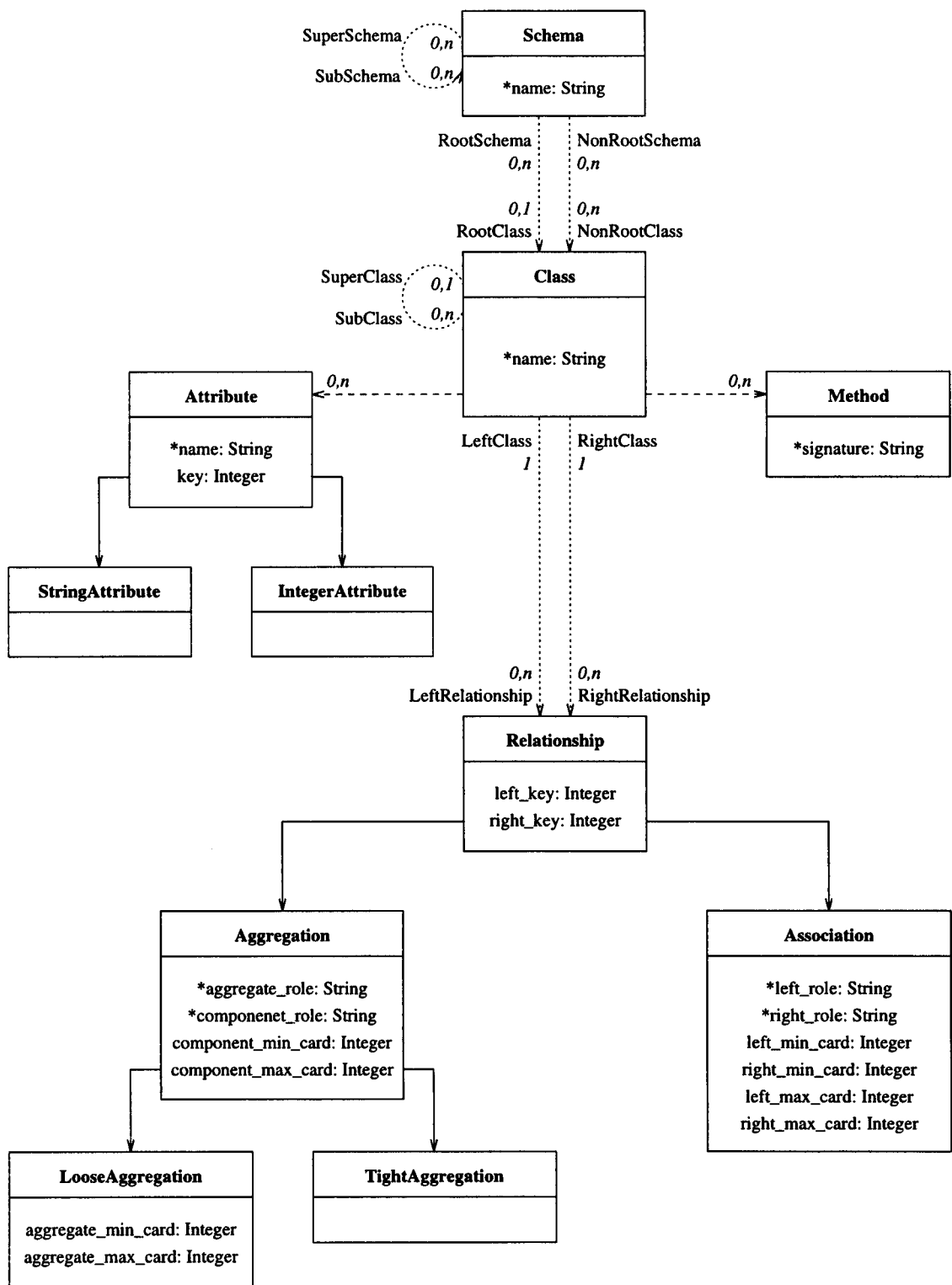


Figure 6.1 The meta-schema

tively. Since the meta-class **Attribute** is superclass of the meta-class **StringAttribute** then o_5 must be related to o_6 : the class o_5 is superclass of the class o_6 .

$$className(o_5) = \text{Class} \quad o_5 \dashrightarrow \text{name} = \text{Attribute}$$

$$className(o_6) = \text{Class} \quad o_6 \dashrightarrow \text{name} = \text{StringAttribute}$$

$$o_5 \begin{array}{c} \xrightarrow{\text{SubClass}} \\ \xleftarrow{\text{SuperClass}} \end{array} o_6$$

Naming Assumptions

All class and schema names used in the meta-schema are considered as “reserved”, i.e., user-defined schemas cannot contain those names. (In Chapter 7 we relax this constraint by introducing *contexts* as a means of defining autonomous name spaces.) Moreover, the only primary types used in the schema correspond to the domains of integers and strings. Thus, let us assume the existence of the following sets:

- a set \mathcal{R}_{CN} of reserved class names:

$$\mathcal{R}_{CN} = \{ \\ \text{Class, Attribute, Method, Relationship, Schema,} \\ \text{IntegerAttribute, StringAttribute,} \\ \text{Aggregation, LooseAggregation, TightAggregation, Association} \\ \}$$

- a set \mathcal{R}_{WN} of reserved schema names:

$$\mathcal{R}_{WN} = \{\text{Meta, Class, Attribute, Method, Relationship, Schema}\}$$

- a set \mathcal{R}_{PN} of reserved primary type names:

$$\mathcal{R}_{PN} = \{\text{Integer}, \text{String}\}$$

Meta-classes

Now we formalise the definitions of meta-class and meta-object, and we postulate the existence of the meta-classes. The meta-classes are the predefined classes of the meta-schema. Thus, the name of a meta-class must be one of the reserved class names. Since meta-classes are predefined they must exist in the set of all classes. A meta-object is any instance of a meta-class, i.e., the most-specific class of a meta-object is a meta-class.

Definition 6.1 (Meta-class) A class $c \in \mathcal{C}^3$ is a meta-class iff $\text{className}(c) \in \mathcal{R}_{CN}$.

□

Invariant 6.1 (Meta-classes Existence) $\forall n \in \mathcal{R}_{CN} : \exists c \in \mathcal{C}$ such that $\text{className}(c) = n$.

◆

Definition 6.2 (Meta-object) An object $o \in \mathcal{O}^4$ is a meta-object iff $\text{objClassName}(o) \in \mathcal{R}_{CN}$.

□

The meta-classes are formally defined in Appendix C according to the meta-schema depicted in Figure 6.1. Examples of instances of the meta-classes, i.e., meta-objects, are given in Section 6.2, where rules for mapping classes, attributes, methods, relationships, and schemas into meta-objects are given.

³The symbol \mathcal{C} is defined by Notation 5.3 as the set of all classes.

⁴The symbol \mathcal{O} is defined by Notation 5.12 as the set of all objects.

Meta-schema

The meta-schema is the schema that designates all meta-classes. As the diagrammatical representation of the meta-schema shows (Figure 6.1), the meta-schema is a super-schema composed of five basic schemas since there are five root classes (Class, Attribute, Method, Relationship, Schema). For simplicity, we name the meta-schema as **Meta** and each sub-schema with the same name of its root class. Moreover, the meta-schema is self-contained since the set of all meta-classes is totally self-contained, i.e., all class relationships are “within” the meta-schema. Hence, according to Definition 5.6, the meta-schema is formally defined as follows.

Definition 6.3 (Meta-schema) *The meta-schema is a schema $w \in \mathcal{W}^5$ such that:*

1. $w.n = \mathbf{Meta}$
2. $w.H.V = \emptyset \ (\Rightarrow w.H.A = \emptyset)$
3. $w.S = \{s_1, s_2, s_3, s_4, s_5\}$, where:
 - (a) $s_1.n = \mathbf{Class}$
 - (b) $s_1.H.V = \{c \in \mathcal{C} \mid c \leq_\tau \text{class}(\mathbf{Class})\}$
 - (c) $s_2.n = \mathbf{Attribute}$
 - (d) $s_2.H.V = \{c \in \mathcal{C} \mid c \leq_\tau \text{class}(\mathbf{Attribute})\}$
 - (e) $s_3.n = \mathbf{Method}$
 - (f) $s_3.H.V = \{c \in \mathcal{C} \mid c \leq_\tau \text{class}(\mathbf{Method})\}$
 - (g) $s_4.n = \mathbf{Relationship}$

⁵The symbol \mathcal{W} is defined by Notation 5.7 as the set of all schemas.

$$(h) \ s_4.H.V = \{c \in \mathcal{C} \mid c \leq_\tau \text{class}(\text{Relationship})\}$$

$$(i) \ s_5.n = \text{Schema}$$

$$(j) \ s_5.H.V = \{c \in \mathcal{C} \mid c \leq_\tau \text{class}(\text{Schema})\}$$

□

Proposition 6.1 (Meta-schema Self Containment) The meta-schema is self-contained.

□

6.2 Meta-object Mapping

In this Section we show how classes, attributes, methods, relationships and schemas are mapped to meta-objects. The meta-schema models a class as an aggregation of attribute specifications, relationship specifications and methods. For this reason, each class maps to an instance of the meta-class **Class** aggregated to instances of the meta-classes **Attribute**, **Relationship** and **Method**. For simplicity of explanation, we introduce the mapping for each type of class component (attribute, method and relationship) separately and then explain how class hierarchies and schemas are mapped. Figure 6.2 shows a simple schema which is used throughout this Section to illustrate such a mapping.

Firstly, we recall that each class has a distinct name. Consequently, the instances of the meta-class **Class** must have the attribute **name** with distinct values. Such instances are simply referred to as *class meta-objects* since they map classes.

Invariant 6.2 (Class Instance Name Distinction) $\forall x, y \in \xi(\text{Class})$: if $x \dashrightarrow \text{name} = y \dashrightarrow \text{name}$ then $x = y$. ♦

Definition 6.4 (Class Meta-object) Given a class $c \in \mathcal{C}$, the class meta-object with respect to c , denoted as $CMO(c)$, is $\beta \in \xi(\text{Class})$ such that $\beta \dashrightarrow \text{name} = \text{className}(c)$. □

Example 6.1 Let c denote the class named **Person**. The notation $CMO(c)$ denotes the instance β of the meta-class **Class** such that $\beta \dashrightarrow \text{name} = \text{Person}$. ◇

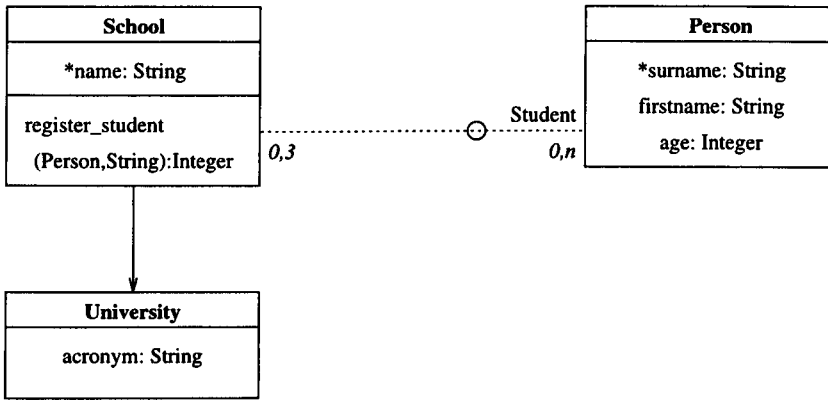


Figure 6.2 Example schema for meta-object mapping

Attribute Mapping

An attribute a of a class c is mapped to two meta-objects: an instance α of the meta-class **Attribute** and an instance β of the meta-class **Class**. The meta-object α must be a direct instance of a subclass of the meta-class **Attribute** (either **StringAttribute** or **IntegerAttribute**) according to the primary type of a . The value of the attribute **name** of α must be the name of a . The meta-object β must be the class meta-object of the class c . The meta-objects α and β must be related: β is the (aggregate) class of α , while α is (component) attribute of β . We introduce

a notation to denote the meta-class corresponding to each primary type and then formalise mapping of attributes to meta-objects.

AttCN

Notation 6.1 Given a primary type name $p \in PN$, the notation $AttCN(p)$ denotes a class name in \mathcal{R}_{CN} as follows.

- $AttCN(Integer) = IntegerAttribute$
- $AttCN(String) = StringAttribute$ □

Meta_A

Definition 6.5 (Attribute Mapping) Given an attribute a of a class c where n is the name of a , p is the primary type name (Integer or String) of a , and k is either 1 or 0 depending whether or not a is a key attribute, the set of meta-objects that maps a , denoted as $Meta_A(a)$, is the set containing only the meta-objects $\alpha \in \xi(Attribute)$, $\beta \in \xi(Class)$ such that:

- (i) $className(\alpha) = AttCN(p)$
- (ii) $\alpha \dashrightarrow name = n$
- (iii) $\alpha \dashrightarrow key = k$
- (iv) $\beta = CMO(c)$
- (v) $\alpha \xrightleftharpoons[Attribute]{Class} \beta$ □

Example 6.2 Let us consider the attribute surname of class Person. Let a denote that attribute, then $Meta_A(a) = \{\alpha, \beta\}$ such that:

$$\begin{array}{ll}
 className(\alpha) = StringAttribute & \beta = CMO(class(Person)) \\
 \alpha \dashrightarrow name = surname & \alpha \xrightleftharpoons[Attribute]{Class} \beta \\
 \alpha \dashrightarrow key = 1 &
 \end{array}$$

Method Mapping

A method m of a class c is mapped to two meta-objects: an instance μ of the meta-class **Method** and an instance β of the meta-class **Class**. The value of the attribute **signature** of μ must correspond to the signature of m . The meta-object β must be the class meta-object of the class c . The meta-objects μ and β must be related: β is the (aggregate) class of μ , while μ is (component) method of β . For practical reasons, the function and the semantics of a method are not mapped to meta-objects. We introduce a notation to denote a string that corresponds to the signature of a method and then formalise mapping of methods to meta-objects.

Notation 6.2 Given a method μ , the notation $StrSig(\mu)$ denotes the string obtained by concatenating the name of μ , the argument type names of μ and the result type name of μ in this order and separating them using commas. □

Definition 6.6 (Method Mapping) Given a method m of a class c , the set of meta-objects that maps m , denoted as $Meta_M(m)$, is the set containing only the meta-objects $\mu \in \xi(\mathbf{Method})$, $\beta \in \xi(\mathbf{Class})$ such that:

- (i) $\mu \dashrightarrow \text{signature} = StrSig(m)$
- (ii) $\beta = CMO(c)$
- (iii) $\mu \overset{\text{Class}}{\underset{\text{Method}}{=}} \beta$

□

StrSig**Meta_M**

Example 6.3 Let us consider the method `register_student` of class `School`. Let m denote that method, then $Meta_M(m) = \{\mu, \beta\}$ such that:

$$className(\mu) = \text{Method}$$

$$\mu \dashrightarrow \text{signature} = \text{"register_student, Person, String, Integer"}$$

$$\beta = CMO(class(School))$$

$$\mu \overset{\text{Class}}{\underset{\text{Method}}{=}} \beta$$

Relationship Mapping

A relationship r between classes c_1 and c_2 is mapped to three meta-objects: an instance σ of the meta-class `Relationship` and two instances β_1, β_2 (which can be the same) of the meta-class `Class`. The meta-object σ must be a direct instance of a subclass of the meta-class `Relationship` (either `TightAggregation`, `LooseAggregation` or `Association`) according to the relationship type. The meta-objects β_1 and β_2 must correspond to the related classes c_1 and c_2 .

Since a relationship is between two classes, to avoid confusion, we must designate one class as the `LeftClass` and the other one as the `RightClass` in the relationship. For simplicity of explanation, we establish a convention where by the class c_1 is the class designated as `LeftClass`, while the class c_2 is the class designated as `RightClass`. Thus, β_1 and σ must be related to each other having, respectively, the roles `LeftClass` and `LeftRelationship`, while β_2 and σ must be related to each other having, respectively, the roles `RightClass` and `RightRelationship`. Moreover, we establish another convention to designate aggregated classes: if the relationship is an aggregation then the aggregate class must be the `LeftClass`, while the

component class must be the **RightClass**. If the relationship is an association then it is unimportant how the classes are designated.

Definition 6.7 (Relationship Mapping) Given a relationship r between classes c_1 and c_2 where

Meta_R

- the role of c_1 is R_1
- the role of c_2 is R_2
- the minimum cardinality of c_1 is l_1
- the minimum cardinality of c_2 is l_2
- the maximum cardinality of c_1 is u_1
- the maximum cardinality of c_2 is u_2
- the flag key of c_1 is k_1
- the flag key of c_2 is k_2

the set of meta-objects that maps r , denoted as $Meta_R(r)$, is the set containing only the meta-objects $\sigma \in \xi(\text{Relationship})$, $\beta_1 \in \xi(\text{Class})$, $\beta_2 \in \xi(\text{Class})$, such that:

- (i) $\beta_1 \xrightleftharpoons[\text{LeftClass}]{\text{LeftRelationship}} \sigma$
- (ii) $\beta_2 \xrightleftharpoons[\text{RightClass}]{\text{RightRelationship}} \sigma$
- (iii) if $relType(r) = \text{"Loose Aggregation"}$ then:
 - (a) $className(\sigma) = \text{LooseAggregation}$
 - (b) $\sigma \dashrightarrow \text{left_key} = k_1$
 - (c) $\sigma \dashrightarrow \text{right_key} = k_2$

-
- (d) $\sigma \dashrightarrow \text{aggregate_role} = R_1$
 - (e) $\sigma \dashrightarrow \text{component_role} = R_2$
 - (f) $\sigma \dashrightarrow \text{component_min_card} = l_2$
 - (g) $\sigma \dashrightarrow \text{component_max_card} = u_2$
 - (h) $\sigma \dashrightarrow \text{aggregate_min_card} = l_1$
 - (i) $\sigma \dashrightarrow \text{aggregate_max_card} = u_1$
- (iv) if $\text{relType}(r) = \text{"Tight Aggregation"}$ then:
- (a) $\text{className}(\sigma) = \text{TightAggregation}$
 - (b) $\sigma \dashrightarrow \text{left_key} = k_1$
 - (c) $\sigma \dashrightarrow \text{right_key} = k_2$
 - (d) $\sigma \dashrightarrow \text{aggregate_role} = R_1$
 - (e) $\sigma \dashrightarrow \text{component_role} = R_2$
 - (f) $\sigma \dashrightarrow \text{component_min_card} = l_2$
 - (g) $\sigma \dashrightarrow \text{component_max_card} = u_2$
- (v) if $\text{relType}(r) = \text{"Association"}$ then:
- (a) $\text{className}(\sigma) = \text{Association}$
 - (b) $\sigma \dashrightarrow \text{left_key} = k_1$
 - (c) $\sigma \dashrightarrow \text{right_key} = k_2$
 - (d) $\sigma \dashrightarrow \text{left_role} = R_1$
 - (e) $\sigma \dashrightarrow \text{right_role} = R_2$
-

$$(f) \sigma \dashrightarrow \text{right_min_card} = l_2$$

$$(g) \sigma \dashrightarrow \text{right_max_card} = u_2$$

$$(h) \sigma \dashrightarrow \text{left_min_card} = l_1$$

$$(i) \sigma \dashrightarrow \text{left_max_card} = u_1$$

□

Example 6.4 Let us consider the association between classes *School* and *Person*. Let the class *School* be designated as the *LeftClass* while the class *Person* as the *RightClass* in that association. Now let r denote that association, then $\text{Meta}_R(r) = \{\sigma, \beta_1, \beta_2\}$ such that:

$$\text{className}(\sigma) = \text{Association}$$

$$\beta_1 = \text{CMO}(\text{class}(\text{School}))$$

$$\beta_2 = \text{CMO}(\text{class}(\text{Person}))$$

$$\beta_1 \xrightleftharpoons[\text{LeftClass}]{\text{LeftRelationship}} \sigma$$

$$\beta_2 \xrightleftharpoons[\text{RightClass}]{\text{RightRelationship}} \sigma$$

$$\sigma \dashrightarrow \text{left_key} = 0$$

$$\sigma \dashrightarrow \text{right_key} = 1$$

$$\sigma \dashrightarrow \text{left_role} = \text{School}$$

$$\sigma \dashrightarrow \text{right_role} = \text{Student}$$

$$\sigma \dashrightarrow \text{left_min_card} = 0$$

$$\sigma \dashrightarrow \text{right_min_card} = 0$$

$$\sigma \dashrightarrow \text{left_max_card} = 3$$

$$\sigma \dashrightarrow \text{right_max_card} = n$$

Class Mapping

A class c is mapped to a set of meta-objects: the union of the sets of meta-objects that map all attributes, all relationships and all methods of β . As a consequence, such a set of meta-objects includes the class meta-object of c and the class meta-objects of all (direct and indirect) superclasses of c . These class meta-objects, in addition, must be related in such way to represent the class

path of c ,⁶ thereby permitting to navigate through meta-objects. Thus, for every pair of classes which have a direct inheritance relation, the corresponding class meta-objects are accordingly related as **SuperClass** and **SubClass**.

Meta_C

Definition 6.8 (Class Mapping) The set of meta-objects that maps a class c , denoted as $Meta_C(c)$, is the set given by:

$$Meta_C(c) = \left(\bigcup_{a \in Att(c)} Meta_A(a) \right) \cup \left(\bigcup_{m \in Met(c)} Meta_M(m) \right) \cup \left(\bigcup_{r \in Rel(c)} Meta_R(r) \right)$$

Proposition 6.2 (Class Path Mapping) $\forall d \in \mathcal{C} : \forall b \in \mathcal{C} : \text{if } d \leq_\tau b \text{ then } CMO(b) \in Meta_C(d)$. □

Invariant 6.3 (Inheritance Mapping) $\forall b, d \in \mathcal{C} : \text{if } b = \text{superclass}(d) \text{ then:}$

$$CMO(d) \xrightleftharpoons[\text{SubClass}]{\text{SuperClass}} CMO(b)$$

Example 6.5 Let us consider the class **University**. Let c denote that class, then:

$$Att(c) = \{a_1, a_2\} \qquad Met(c) = \{m\} \qquad Rel(c) = \{r\}$$

where:

- a_1 is the attribute whose name is **name**
- a_2 is the attribute whose name is **acronym**
- m is the method whose name is **register_student**
- r is the relationship between classes **School** and **Person**

⁶The class path of a class c is the sequence of classes including c all its superclasses.

Thus, from Definition 6.8 (Class Mapping), we have that:

$$Meta_C(c) = \left(Meta_A(a_1) \cup Meta_A(a_2) \right) \cup \left(Meta_M(m) \right) \cup \left(Meta_R(r) \right)$$

Now, let us show that Proposition 6.2 holds. Since class **School** is the only super-class of c (class **University**), we have the following is-a relations for c :

$$c \leq_{\tau} class(School)$$

$$c \leq_{\tau} class(University)$$

From Definition 6.5 (Attribute Mapping), Definition 6.6 (Method Mapping) and Definition 6.7 (Relationship Mapping), we have that:

$$Meta_A(a_1) \supset \{ CMO(class(School)) \}$$

$$Meta_M(m) \supset \{ CMO(class(School)) \}$$

$$Meta_A(a_2) \supset \{ CMO(class(University)) \}$$

$$Meta_R(r) \supset \{ CMO(class(School)), CMO(class(Person)) \}$$

Hence:

$$Meta_C(c) \supset \{ CMO(class(School)), CMO(class(University)) \}$$

which means that the Proposition holds.

Moreover, since class **School** is direct superclass of c ($superclass(c) = class(School)$), according to Invariant 6.3, we have that:

$$CMO(class(University)) \overset{\text{SuperClass}}{\underset{\text{SubClass}}{=}} CMO(class(School))$$

Schema Mapping

Now we show how a schema (including all classes that it designates) is mapped to meta-objects. Firstly, for each schema there must be an instance of the meta-class **Schema**. We recall that, according to Invariant 5.1, each schema has a distinct name. Consequently, the instances of the meta-class **Schema** must have the attribute **name** with distinct values. Such instances are simply referred to as *schema meta-objects* since they map schemas.

Invariant 6.4 (Schema Instance Name Distinction) $\forall x, y \in \xi(\text{Schema})$: if $x \dashrightarrow \text{name} = y \dashrightarrow \text{name}$ then $x = y$. ◆

Definition 6.9 (Schema Meta-object) Given a schema $w \in \mathcal{W}$, the schema meta-object with respect to w , denoted as $\text{SMO}(w)$, is $s \in \xi(\text{Schema})$ such that $s \dashrightarrow \text{name} = \text{schemaName}(w)$. □

Example 6.6 Let us consider the schema depicted in Figure 6.2. For simplicity of notation, let us denote the schema by w_3 , and let us suppose that its name is **Academia** ($\text{schemaName}(w_3) = \text{Academia}$). Thus, the notation $\text{SMO}(w_3)$ denotes the instance s of the meta-class **Schema** such that $s \dashrightarrow \text{name} = \text{Academia}$. ◇

Secondly, a schema w is composed of a root-subtree and a set of sub-schemas.⁷ Hence, the set of meta-objects that maps w includes the schema meta-object of w , all meta-objects that map the classes pertaining to the root-subtree of w and, recursively, all meta-objects that map the sub-schemas of w .

⁷Formally, (Definition 5.6) a schema w consists of a root-subtree whose set of classes is denoted $w.H.V$ and a set of sub-schemas denote as $w.S$.

Definition 6.10 (Schema Mapping) The set of meta-objects that maps a schema w , denoted as $Meta_W(w)$, is the set given by:

 $Meta_W$

$$Meta_W(w) = \{SMO(w)\} \cup \left(\bigcup_{\beta \in w.H.V} Meta_C(\beta) \right) \cup \left(\bigcup_{x \in w.S} Meta_W(x) \right)$$

Therefore, given a schema meta-object s that is the schema meta-object of a schema w , it should be possible to navigate through all meta-objects that map the root-subtree of w . For this reason, the class meta-objects corresponding to the classes pertaining to the root-subtree of w must be related to s in such way to reflect whether the class is or is not the root of the root-subtree.

Invariant 6.5 (Root-subtree Mapping) $\forall w \in \mathcal{W}$:

- (i) $\forall c \in w.H.V$: if $c = Root(w.H)$ then $CMO(c) \xrightleftharpoons[\text{RootClass}]{\text{RootSchema}} SMO(w)$
- (ii) $\forall c \in w.H.V$: if $c \neq Root(w.H)$ then $CMO(c) \xrightleftharpoons[\text{NonRootClass}]{\text{NonRootSchema}} SMO(w)$ \blacklozenge

Example 6.7 Let us consider the schema depicted in Figure 6.2. Since there are two root classes (School and Person) the schema is a super-schema composed of two basic schemas. For simplicity of notation, let us denote the basic schema rooted at class School by w_1 and the basic schema rooted at class Person by w_2 .

Thus, we have that:

$$\begin{aligned} CMO(class(School)) &\xrightleftharpoons[\text{RootClass}]{\text{RootSchema}} SMO(w_1) \\ CMO(class(Person)) &\xrightleftharpoons[\text{RootClass}]{\text{RootSchema}} SMO(w_2) \\ CMO(class(University)) &\xrightleftharpoons[\text{NonRootClass}]{\text{NonRootSchema}} SMO(w_1) \end{aligned}$$

Also, it should be possible to navigate through all meta-objects that map the sub-schemas of a schema. Thus, for every sub-schema s of a schema w there

must be a relationship between the schema meta-object of w and the schema meta-object of s to reflect the nesting of schemas.

Invariant 6.6 (Schema Nesting Mapping) $\forall w, s \in \mathcal{W}$: if $s \in w.S$ then:

$$SMO(s) \overset{\text{SuperSchema}}{\underset{\text{SubSchema}}{\rightleftharpoons}} SMO(w)$$

Example 6.8 Let us consider the super-schema depicted in Figure 6.2. Let us denote the sub-schema rooted at class **School** by w_1 , the sub-schema rooted at class **Person** by w_2 , and the super-schema by w_3 . Thus, we have that:

$$SMO(w_1) \overset{\text{SuperSchema}}{\underset{\text{SubSchema}}{\rightleftharpoons}} SMO(w_3) \quad SMO(w_2) \overset{\text{SuperSchema}}{\underset{\text{SubSchema}}{\rightleftharpoons}} SMO(w_3)$$

6.3 Summary

We summarise the discussion on meta-object mapping by presenting, as an example, the complete set of meta-objects that maps the schema depicted in Figure 6.2. Firstly we identify all elements in the schema. As shown in Figure 6.3, where the nesting of schemas is made explicit, attributes are denoted by a_1, \dots, a_5 , the only method is denoted by m_1 , the only relationship is denoted by r_1 , classes are denoted by c_1, c_2, c_3 , root-subtrees are denoted by H_1, H_2 , and schemas are denoted by w_1, w_2, w_3 .

The set of meta-objects that map all schema elements is diagrammatically represented in Figure 6.4. Each meta-object is represented by a rectangle with rounded corners and has four parts: the two top parts contain the object class name (c) and the object name (n), the intermediary part contains the object

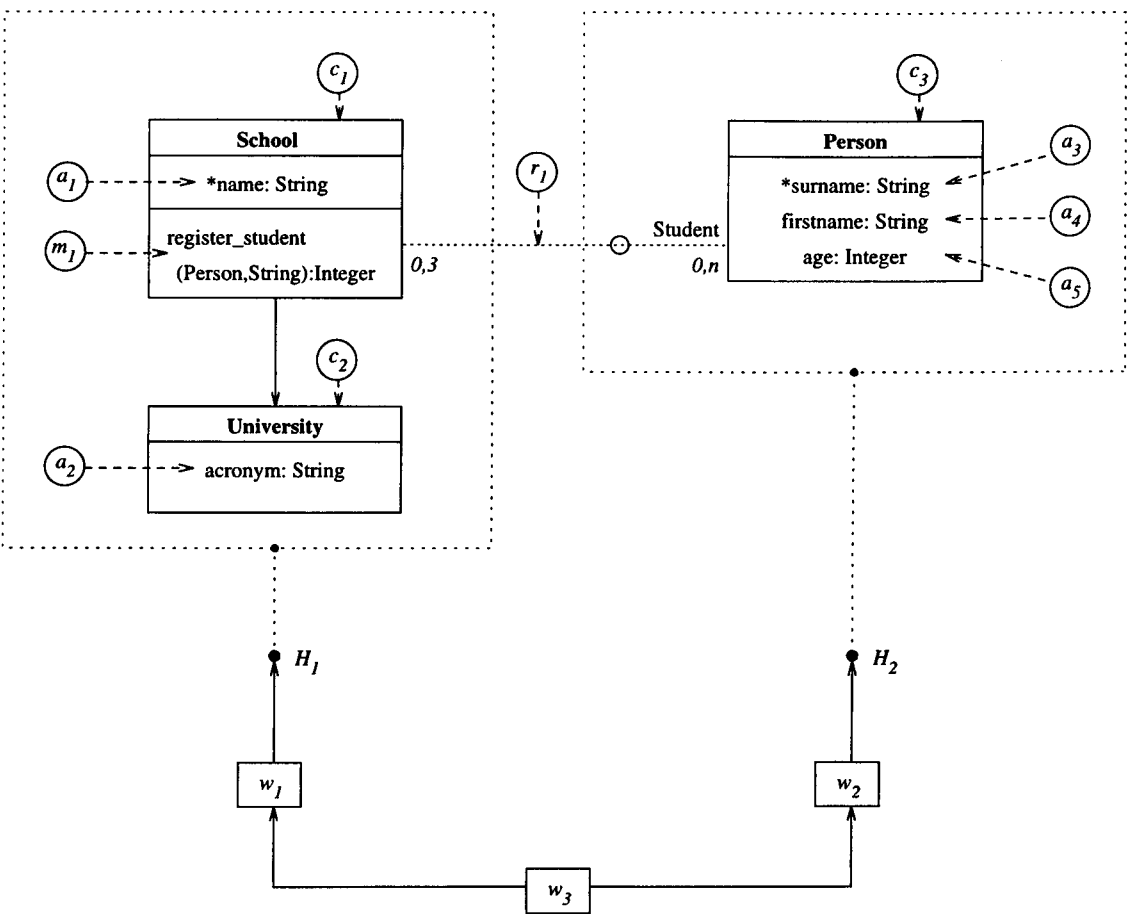


Figure 6.3 Example schema with annotated elements

attributes, and the bottom part contains the object relationships. For simplicity, objects are named $1, \dots, 13$, and each relationship is designated by the name of the corresponding related role. Although object relationships are already shown through object names in reference sets of relationship variables, a double-headed arrow is present between two related objects to emphasise their relationship.

The mapping of all schema elements into meta-objects is formally given in Appendix D. Figure 6.5 helps to visualise the sets of meta-objects that map each schema element. Objects are simply represented by circles containing the

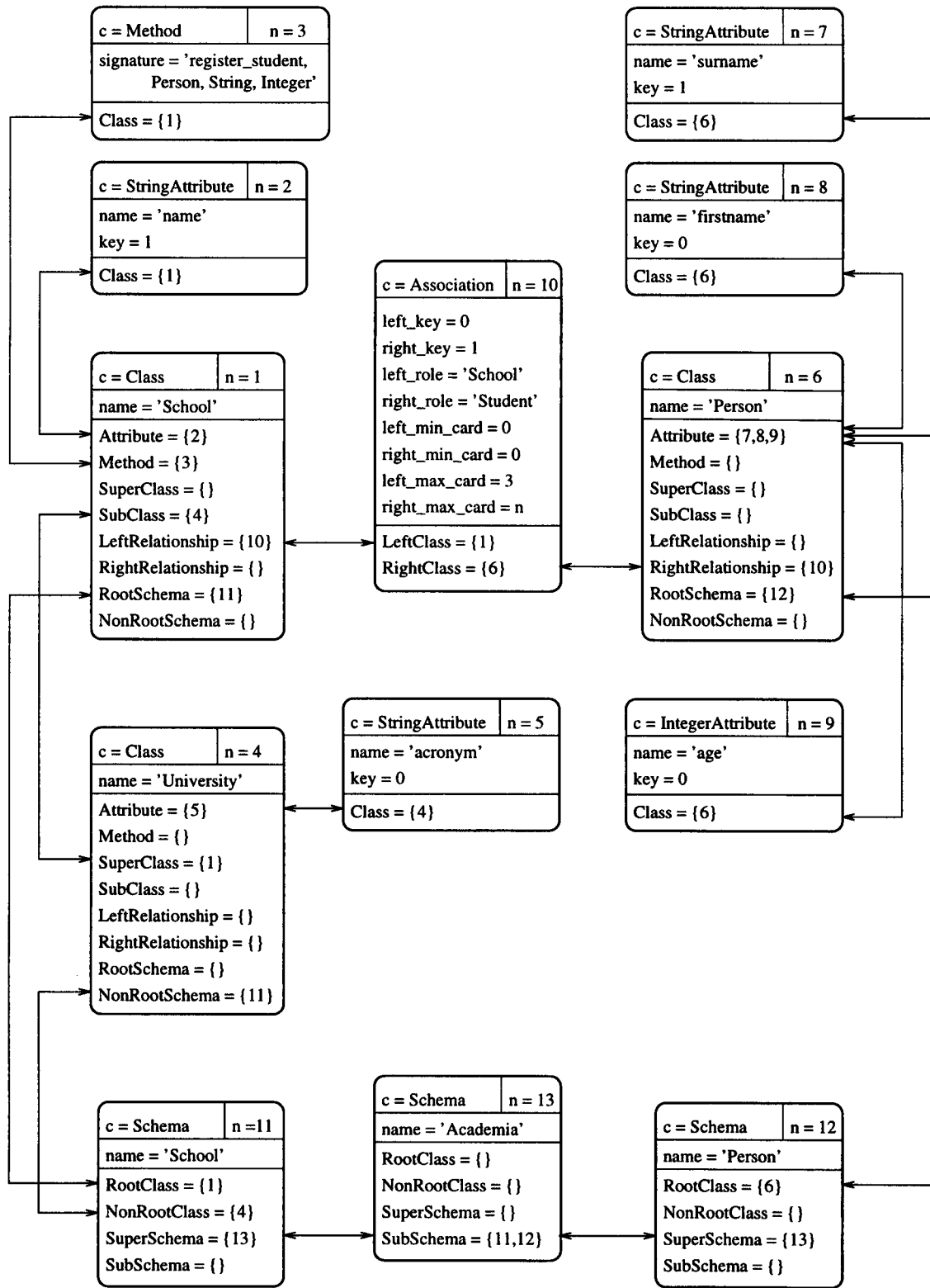


Figure 6.4 Meta-objects for schema in Figure 6.3

corresponding object name $(1, \dots, 13)$, and each set is surrounded by a polygon. We use different line styles for the sake of clarity only; there is no specific meaning for each line style. Also for simplicity, each set is named with the same name of the schema element mapped by the set (a_1, a_2, etc) .

6.4 Conclusions

Although simple the meta-object model permits complete and unambiguous representation of schemas. This enables the use of meta-objects for several purposes, including schema management, system documentation, software management, implementation of the type space and, as a novelty, a simple implementation of query interpreters.

The uniform representation of information *modelled by* schemas (instances of classes) and the information *pertaining to* schemas (class definitions) permits manipulation of information and meta-information to be done in a uniform and integrated way, thereby greatly simplifying the system interface. For example, users can navigate between information and meta-information using a single interface; an information browser or query interpreter can operate on both information and meta-information.

Finally, the ability of the meta-schema in representing itself makes the system architecture very concise and independent of other information models.

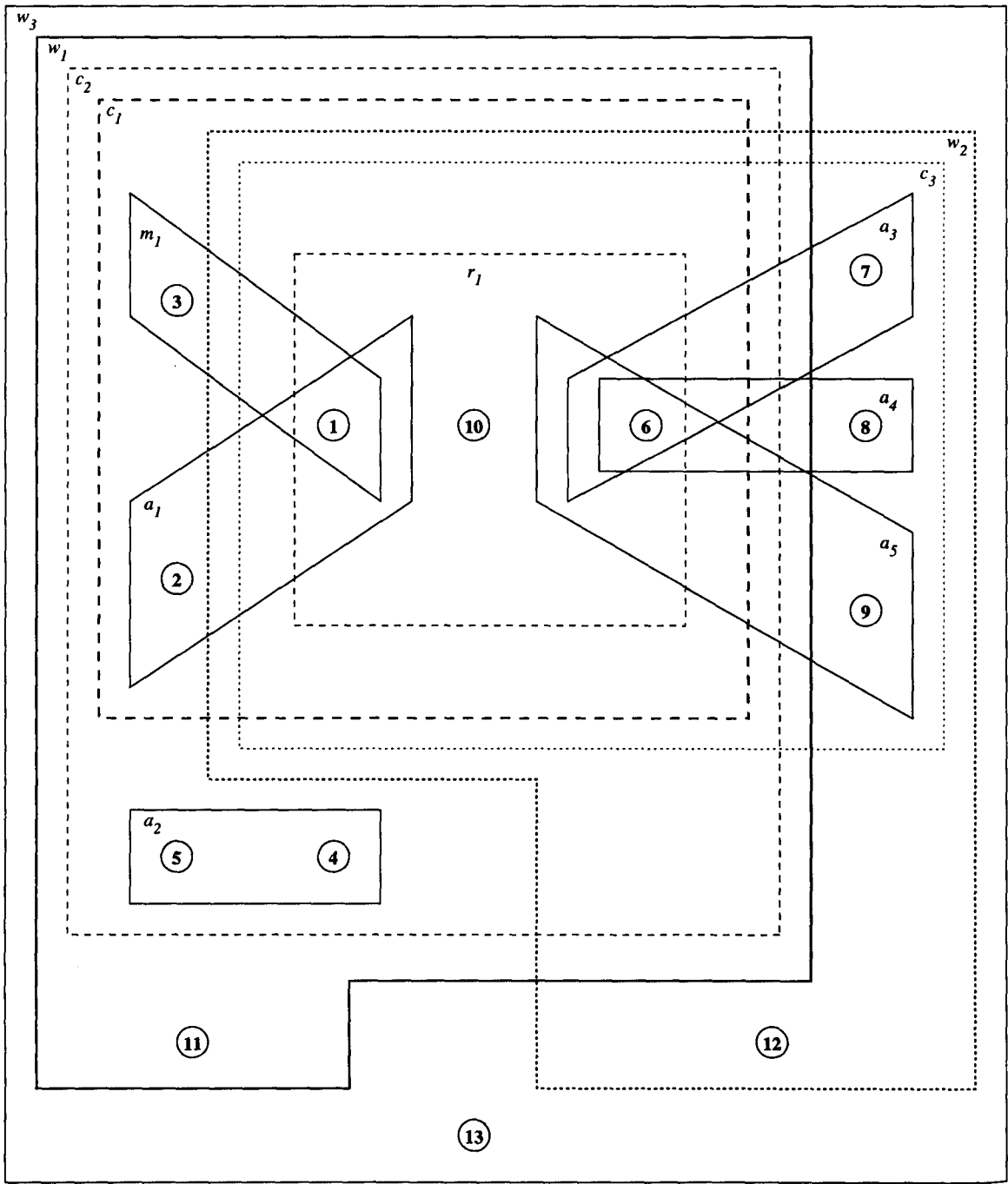


Figure 6.5 Sets of mapping meta-objects in Figure 6.4

CHAPTER 7

Object Space Organisation

In this Chapter we complete the description of object engine components by firstly defining the index information that should be maintained about objects, secondly defining *views* as a means of grouping corresponding meta-objects, objects and indices, and thirdly defining *contexts* as repositories for views, i.e., repositories for meta-objects, objects and indices. All the definitions presented in this Chapter are also formally presented in Appendix E.

7.1 Indices

As discussed in Chapter 3, object engines maintain two types of indices: attribute indices and relationship indices. Let us define indices using generic classes and corresponding instances, as shown in Figure 7.1. Firstly, let us consider the class with instances depicted in Figure 7.1(a) and define attribute index. A class C has a key attribute a of a primary type p (e.g. integer or string), and k instances with names c_i having (not necessarily distinct) values v_i , $1 \leq i \leq k$, for the attribute a . Thus, the attribute index with respect to class C and attribute a is the relation

given by the following set of ordered pairs:

$$\{(v_1, c_1), (v_2, c_2), \dots, (v_k, c_k)\}$$

Now, let us consider the classes with instances depicted in Figure 7.1(b) and define relationship index. The classes A and B have a relationship r where the role of A is RA, the role of B is RB, and the relationship is key with respect to both classes. A collection of (not necessarily distinct) k instances of A and a collection of (not necessarily distinct) k instances of B, respectively named a_i and b_i , $1 \leq i \leq k$, are related to each other with respect to the relationship r . Since the relationship r is key with respect to class A, the relationship index with respect to class B and role RA is the relation given by the following set of ordered pairs:

$$\{(a_1, b_1), (a_2, b_2), \dots, (a_k, b_k)\}$$

And, since the relationship r is key with respect to class B, the relationship index with respect to class A and role RB is the relation given by the following set of ordered pairs:

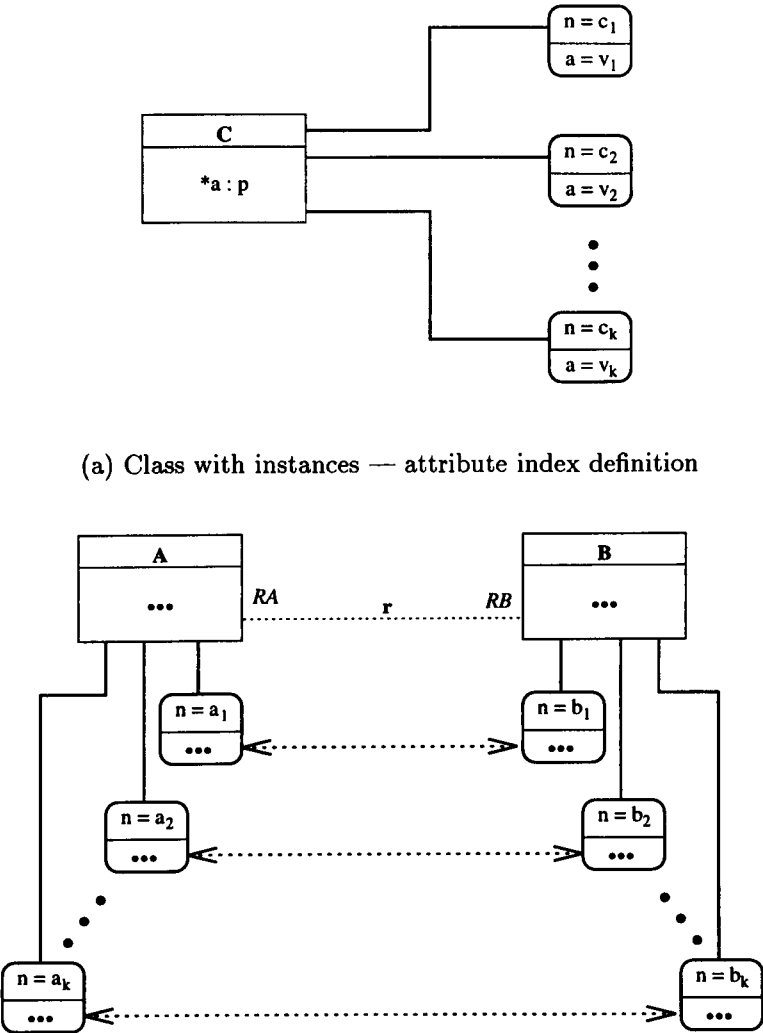
$$\{(b_1, a_1), (b_2, a_2), \dots, (b_k, a_k)\}$$

For simplicity, we call the union of the set of all attribute indices with respect to a class T and attributes of T with the set of all relationship indices with respect to T and roles of classes related to T the *set of indices of T*. Also we introduce a notation to denote the set of all indices of all classes designated by a schema.

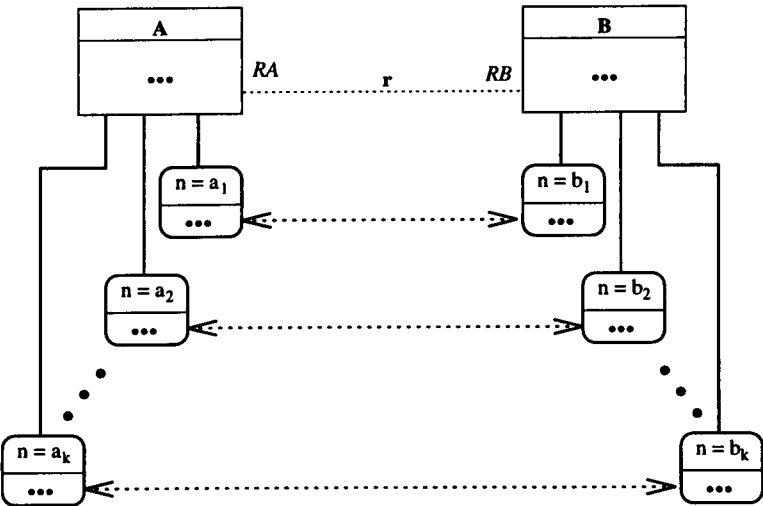
Index_w

Notation 7.1 Given a schema w , the notation $\text{Index}_w(w)$, denotes the set of indices with respect to w , i.e., the set of all indices of all classes designated by w .

□



(a) Class with instances — attribute index definition



(b) Related classes with instances — relationship index definition

Key:

$n = x$

...

An object whose name is x

————

Instance-of relationship

Class relationship

$\leftarrow \cdots \rightarrow$

Object relationship

Figure 7.1 Classes with instances for the definition of indices

7.2 Views

We recall that, in Chapter 5, self-contained schema is defined as a collection of classes that is self-contained with respect to hierarchy and relationship, while database is defined as the set of objects designated by a self-contained schema and, therefore, can be manipulated independently. Also we recall that, in Chapter 6, we defined what a set of meta-objects that map a schema is, i.e., the set of meta-objects that constitute the meta-data corresponding to the classes of a schema. Now, with the definition of indices, we are able to define *views* as entities that designate portions of the information space maintained by an object engine, including the corresponding meta-data and index information as well as the information itself. The view with respect to a self-contained schema w has (1) the same name as w and consists of (2) the database with respect to w , (3) the set of meta-objects that maps w , and (4) the set of indices of w . Formally, the definition of views is given as follows.

Definition 7.1 (View) *Given a self-contained schema $w \in \mathcal{W}$, a view with respect to w is a tuple $(n, \delta, \Pi, \vartheta)$, where:*

- $n \in \mathcal{WN}$
- δ is a database
- Π is a set of meta-objects
- ϑ is a set of indices

such that:

- (i) $n = w.n$

$$(ii) \delta = DB(w)$$

$$(iii) \Pi = Meta_W(w)$$

$$(iv) \vartheta = Index_W(w)$$

□

7.3 Contexts

To reiterate the discussion in Chapter 3, the purpose of contexts is twofold:

1. A context designates an object engine, thereby serving as the starting point for any interaction with client programs. For simplicity, contexts are named globally, i.e., the space of contexts is flat.
2. A context defines an independent name space for primary types, classes schemas and views, i.e., these entities are named within each context independently of any other context.

Thus, the names of primary types, classes, schemas and views are *context relative*, while the names of contexts themselves are global. A formal definition of contexts is presented in Appendix E. In a simple way, a context is a (flat) directory or a container of all views defined for a given object engine. We recall that, as discussed in Chapter 3, the bootstrap of a context generates a special set of meta-objects that represents the meta-data corresponding to the meta-schema, in order to permit object engine administrators to create meta-objects corresponding to user-defined schemas. Accordingly, we call this special set of meta-objects *meta-view* and, for simplicity, name the view **Meta**.

An example of context with views is illustrated in Figure 7.2. The name of the context is **Renoir**. The context contains the predefined view **Meta**, and the user-defined views **Museum**, **Bank** and **Account**. We can note that each view contains three distinct sets: a set of meta-objects, a set of objects (a database) and a set of indices. In particular, due to the reflexive architecture of object engines, the set of meta-objects corresponding to the special view **Meta** is a subset of the corresponding database. Also, we can note that the view **Account** designates a portion of the information space that is a subset of the portion designated by the view **Bank**.

7.4 Conclusions

Views and contexts provide a powerful yet simple framework for organising the object space defined by an object engine. Moreover, since the set of objects designated by views are databases, views can be used to scope the information space manipulated by each client program interaction. As a consequence, views provide an opportunity for efficient implementation of run-time type information necessary by client programs to interact with object engines. Furthermore, the definitions of attribute and relationship indices permit full representation of information necessary to resolve queries, as we will discuss in Chapter 8. Therefore, all components of object engines are harmoniously arranged to deliver the target services.

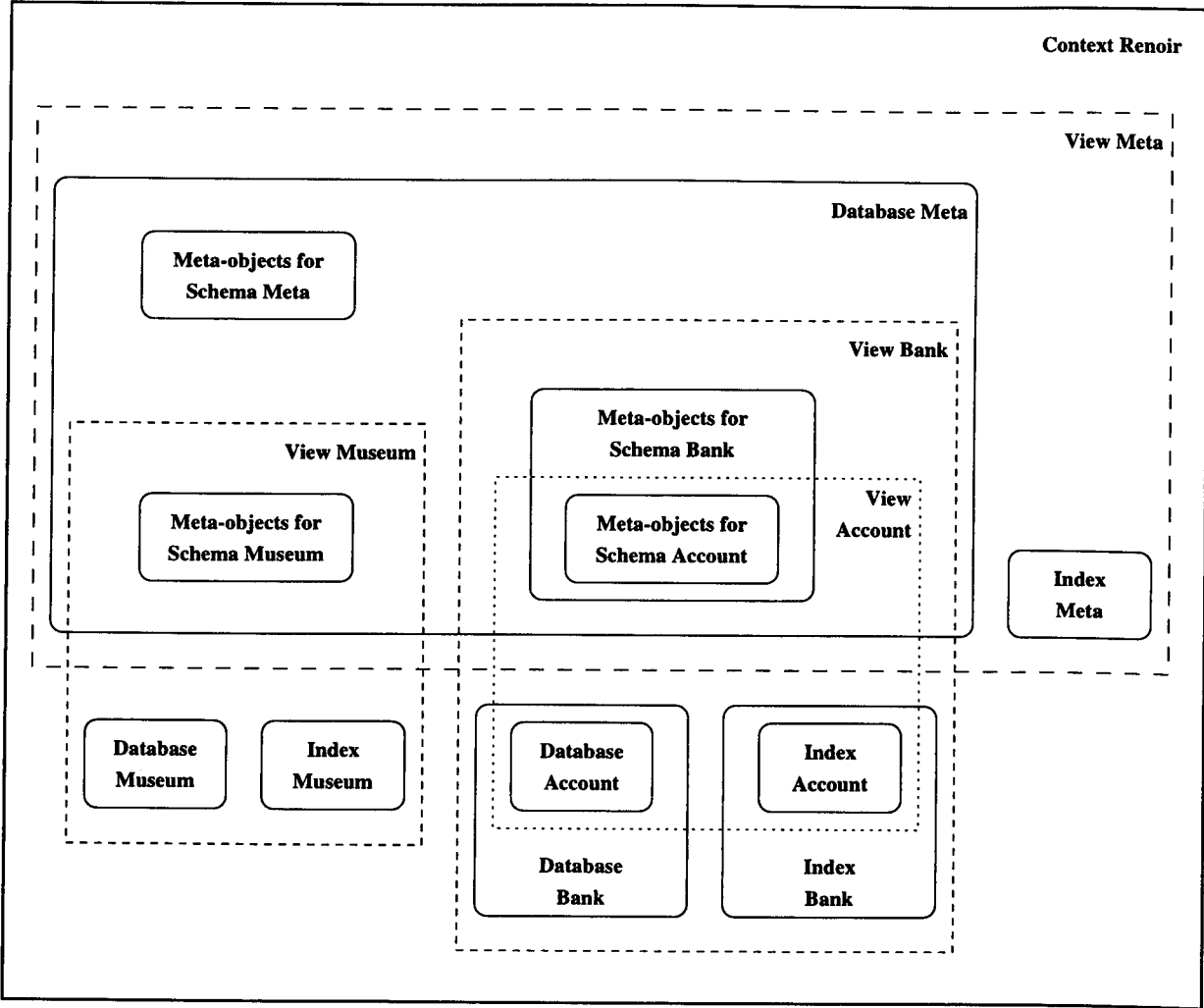


Figure 7.2 Example of context with views

CHAPTER 8

Stabilis Toolkit

In this Chapter we describe the design and implementation of a toolkit for constructing object engines named **Stabilis**. Besides implementing all the definitions we have developed so far, Stabilis provides an object query language and the necessary support for distributed manipulation of objects by programs. Stabilis is implemented as an extensible class library atop the Arjuna system (Section 2.4).

Before we describe Stabilis, let us clarify a point in our notation. Normally, in homogeneous object-oriented systems, all components are implemented as objects. Moreover, all interactions of client programs are with (language) objects. However, until now we have used the word *object* specifically to designate the architectural components of object engines which are extracted from network resources and, obviously, this can cause some confusion in our discussion. Since this Chapter is concerned with systems implementation and the word object appears very frequently with its general meaning, we refer to objects having the meaning of architectural components of object engines as *user objects* when a distinction is necessary. This notation also differentiates the components which are internal to the implementation of object engines from the external ones, i.e., the user objects.

8.1 Implementational Components

The implementational components of object engines provided by Stabilis are depicted in Figure 8.1. The black box represents a client program. The components **Context**, **View** and **Index** correspond to the architectural components of object engines named in the same way. The component **User Object** corresponds to object engines' objects (which can be meta-objects in the case where the client program is an administrator). The remaining components are introduced to provide support for distribution, and they will be explained throughout this Section. An arrow from a component *A* to a component *B* indicates that *A* has a reference to *B* and, therefore, *A* can invoke operations or methods of *B*. If the arrow is shown with solid line then the invocation must be local, else, if the arrow is shown with dashed line then the invocation can be remote (RPC). For more clarity, a thick line surrounding a set of components indicates that the components are co-located, i.e., they are located in the same address space or node, and, consequently, only local invocations can happen between the components. Accordingly, we refer to the node where the client program (represented by the black box) resides as the *client node*. We can observe that the client program, contexts, views and user objects are *conceptually* co-located (in the client node), while indices are remotely accessed by views. The actual physical distribution of these components as well as the reasons for the different arrangements will be explained below.

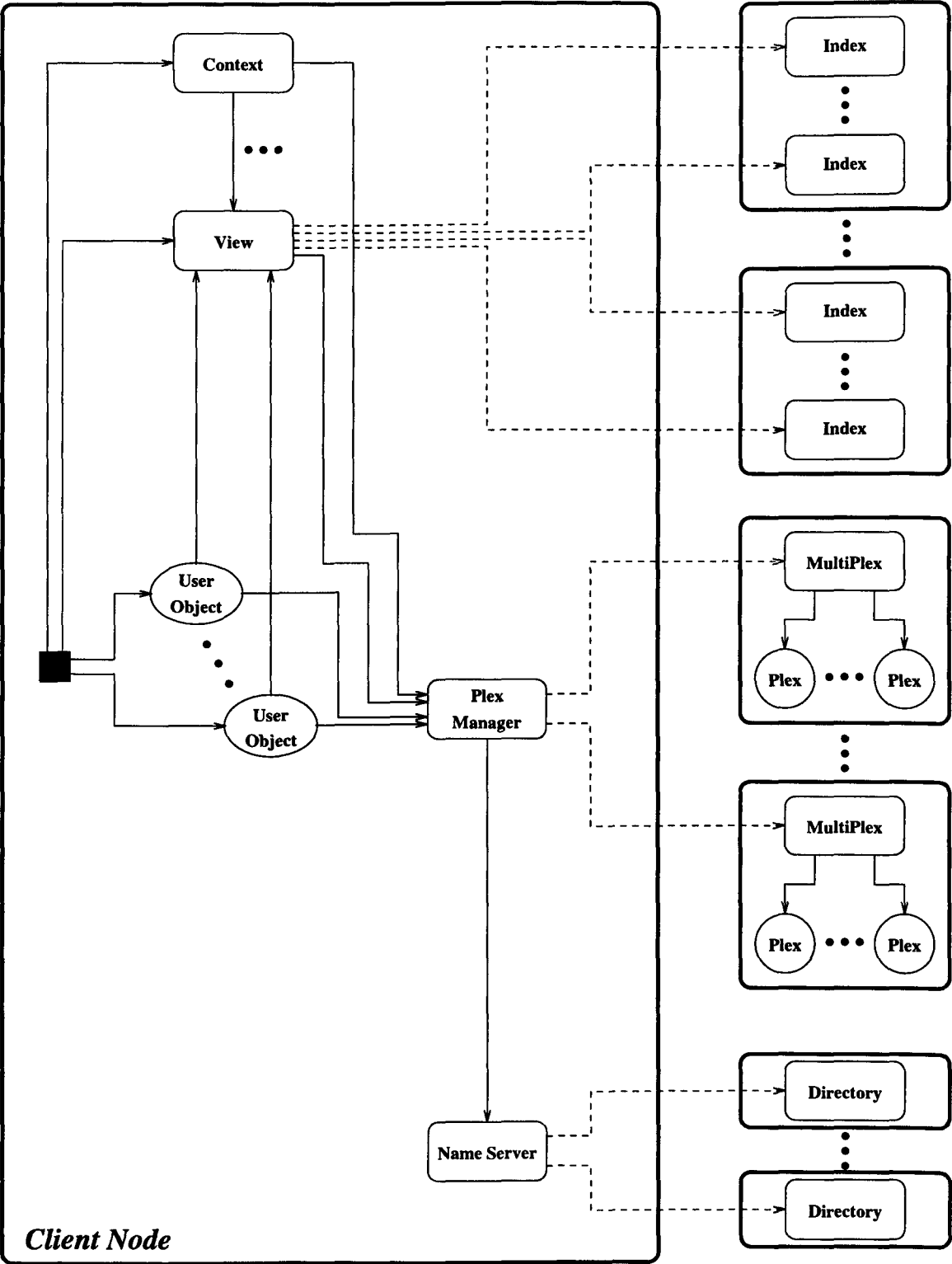


Figure 8.1 Implementational components of object engines

Overview

A client program initiates interaction with an object engine by first retrieving an instance of context. Next the client program asks the context to retrieve a specific instance of view and return a reference to it. (The context is specified through its unique name, while the view is specified through its distinct name with respect to the context.) Since a view contains meta-data that represents a self-contained schema, and also references to the corresponding indices, the client program can manipulate the user objects (i.e., the database) designated by the obtained view. Every user object manipulated by the client program receives, at the instantiation time, the reference to the view. Thus, each user object interacts with the view for the purposes of indexing information update and query resolution, as follows. When the client program creates a new user object, the user object itself asks the view to *insert* indexing information. When the client program modifies a user object, the user object itself asks the view to *update* indexing information. When the client program retrieves a user object through a query, the user object itself asks the view to resolve the query (which makes use of indexing information). When the client program deletes a user object, the user object itself asks the view to remove indexing information. The structure of views and indices are described detail in Section 8.1.2, and the operations on user objects are explained in Section 8.2.

Mobile Objects × RPC

The components directly manipulated by a client program are contexts, views and user objects. Although these components or objects are physically distributed, a client program has the illusion of a single, global object space, i.e., all invocations are local. This is accomplished by temporarily *moving*, on demand, the object from the node where it normally resides, i.e., its *home node*, to the client node; the object stays in the client node while the client program manipulates it and, after that, the object returns to its home node. For this reason, we refer to these objects as *mobile objects*. Thus, the methods of a mobile object can be invoked only while the object is located at a client node; a mobile object is a passive entity while located in its home node. Furthermore, mobile objects are persistent, operations on mobile objects are of type *read* or *write*, and mobile objects can be concurrently manipulated, according to the *multiple reads, single write* semantics. That means a mobile object can be present in more than one client node simultaneously, when all the client programs invoke only read operations of the mobile object. Contexts, views and user objects are implemented as mobile objects, rather than as remote objects accessed through an RPC mechanism, for the following reasons:

1. Contexts are relatively small objects, they suffer few modifications during their lifetime, client programs manipulate few of them, and client programs sporadically invokes read-only operations (client programs simply ask contexts to return views). Thus, contexts can be moved once to a client program and then copied in cache for use by the client program until the end of its execution. If a RPC mechanism is used instead, either a server process for each context manipulated by a client program is active while the client pro-
-

gram may want to retrieve views or a new server process is created every time a client program really wants to retrieve a view. In both alternatives, there is an overhead in process management.

2. Views are relatively small objects, they suffer few modifications during their life time, client programs manipulate few views, and client programs invoke only their read operations (index information update and query resolution) but with great frequency. Thus, views can be moved once to a client program and then copied in cache for use by the client program until the end of its execution. If a RPC mechanism is used instead, all operations would be remote, causing communications overhead and delay in the manipulation of user objects.
 3. User objects are expected to be relatively small, client programs are expected to manipulate user objects of many different classes, they typically provide methods which are small and very often invoked by client programs (attribute update and display, relationship creation, deletion and traversal, user-defined method invocations), very often they are meant to be *fully* displayed to users, and they are expected to be highly concurrently manipulated. Thus, user objects can be moved to client programs every time there is an atomic computation to be performed on them (which may encompass several method invocations). If a RPC mechanism is used instead, firstly the suite of different server processes necessary to accommodate the number of different classes of user objects would be prohibitive in terms of administration, disk space and simultaneously active processes, and secondly the large number of method invocations would cause high communications overhead. Moreover, the whole object would have to be inevitably transported by the
-

RPC for its full display to the user.

Indices, on the other hand, are better accommodated by a RPC mechanism, rather than by object mobility. Indices are expected to be large and highly concurrently manipulated objects, and client programs are expected to access many different indices. Moreover, client programs access indices indirectly (client programs invoke operations of views, which then invoke operations of indices). Thus, in terms of location transparency, client programs would not benefit from an object-mobility-based implementation of indices.

Therefore, both object mobility and RPC mechanisms are important for object engines. However, the Arjuna system does not provide object mobility directly. As discussed in Chapter 2, the model of distribution in Arjuna is client-server processes with communication through RPC. For this reason, we implemented an object mobility mechanism atop the Arjuna system.

8.1.1 Object Mobility Mechanism

Each mobile object is implemented by two parts: a *passive* part at its home node and an *active* part at the client node. The passive part corresponds to the *object state*, while the active part corresponds to the *object behaviour*, i.e., the methods that operate on the object state. Thus, when we say that an object moves between its home node and a client node, what actually moves is an object state.

In Figure 8.1, the active parts of mobile objects correspond to the components Context, View and User Object, while the passive parts correspond to the component Plex (indicated by a circle). For each instance of context, view and user

object there is an instance of plex. Although there is no such indication in the Figure, the active part of a mobile object has, in fact, two subparts: a *general* part and *specific* part. As the names suggest, the general part corresponds to behaviour which is common to any mobile object, while the specific part corresponds to behaviour which is application dependent. We refer to the general part as **Object Manager** as its behaviour encompasses the management of the object state at the client node and provides an interface to the specific part that exempts programmers from any detail related to object mobility.

The transport of object states is realised by the component **MultiPlex** which can handle all plexes that reside in a certain node. For each client program, there must exist a multiplex for any node that hosts a plex in use by the client program. The component **Plex Manager** handles all instances of multiplexes in use by a client program and provides an interface that exempts object managers from having to deal with distribution. Finally, the components **NameServer** and **Directory** provide a simple naming service, basically to associate the identification of a plex with the identification of its home node.

Plex

A plex maintains a flat representation of the state of a mobile object, i.e., a plex encapsulates an object state. A plex is implemented as an Arjuna persistent object and, because of it, plexes can be manipulated from any node in the distributed system through RPC and with transactional access (concurrency and recovery control). Also, they can be replicated as necessary to provide for availability and scalability. A plex is simply created by providing an object state as a parameter.

As a consequence, a unique identification (UID) is automatically assigned to it by the Arjuna system; this UID is then used as the identification (global name) of the corresponding mobile object. The interface of a plex contains only basic operations for the manipulation of the object state, including: return the object state and set a (read or write) lock on itself, update the object state to a new value, set a (read or write) lock on itself, and (permanently) destroy itself.

MultiPlex

A multiplex handles the plexes that reside in a certain node and are in use by the corresponding client program. The multiplex operations are remotely invoked using Arjuna's RPC mechanism. These operations include: create a plex giving an object state as a parameter, return the object state maintained by a plex and set a (read or write) lock on the plex, write the new object state of a plex, set a (read or write) lock on the plex, destroy (permanently) a plex, and discard a plex (from the control of the multiplex).

Plex Manager

The plex manager totally conceals multiplexes, i.e., the interface of the plex manager provides operations simply for the manipulation of plexes. Basically, such operations include: create a new plex given an object state, read the object state of a plex and set a (read or write) lock on it, write the new object state of a plex, set a (read or write) lock on a plex, destroy (permanently) a plex, and discard a plex (from the control of the plex manager, when the mobile object is no longer

of interest for the client program). The plex manager is responsible for the placement of objects, i.e., the plex manager decides on which node a plex should be created. Also, the plex manager is responsible for keeping the name server up-to-date, i.e., the plex manager registers every newly created plex with the name server. Although Stabilis provides no mechanism for object migration, systems administrators can use external tools to replace plexes. In this case, the name server must be updated through an appropriate tool provided by Stabilis.

Object Manager

The operations provided by a plex manager must be invoked at the right time and in the right order, according to the semantics of the mobile objects. However, this may be a complex task and error prone, specially when the number of mobile objects is large and involving complex interactions and dependencies. This calls for the provision of a mechanism to manage the invocations of operations of the plex manager properly. Moreover, such a mechanism should provide a simple and safe interface in order to make it easier to program the active part of mobile objects. This is accomplished by the object manager, the general subpart of the active part of each mobile object.

The object manager is implemented by a class named `Object`, and classes whose instances are mobile objects (which includes contexts, views and user objects) must be subclasses of the class `Object`; the (super)class `Object` implements the general behaviour, while the subclasses implement the specific behaviour of mobile objects. For this discussion, we simply refer to any of these subclasses (and, recursively, subclasses of these subclasses) as a *managed class*. Thus, a

managed class inherits methods from the class `Object` which must be invoked in the implementation of the managed class' methods, as discussed below.

Our approach to the implementation of the object manager has its basis in the underlying system. A factor that contributed to the success of the object and action model of computation, in particular as implemented by the Arjuna system, is the simplicity of the programming interface. To reiterate the description presented in Chapter 2, basically, programs are structured as method invocations controlled by nested atomic actions, while the implementation of each user-defined class sets the necessary locks, according to the semantics of the methods. The user classes obtain concurrency control, recovery and persistence mechanisms through inheritance; a user-defined class must be subclass of a standard class `LockManager` and its methods must invoke the inherited method `setlock` with parameter *read* or *write*, in the current implementation of Arjuna.

Furthermore, the programming interface of the object manager harmoniously integrates with the atomic action programming interface provided by Arjuna. As a simple example, let us firstly consider the C++ code shown Figure 8.2 which shows the typical structure of Arjuna user classes' methods. In line 1, an atomic action *A* is initiated. In line 2, the method tries to lock the object for *write*. If the lock is not granted then the execution goes to line 7, where the atomic action *A* is aborted. Otherwise, in line 4, the method modifies object attributes and then, in line 5, ends the atomic action *A*. Now, let us consider the C++ code shown in Figure 8.3 which is the equivalent typical structure of managed classes' methods. The first observation is the introduction of a simple exception handling mechanism. The use of this mechanism is achieved by creating an object of the

class `OpHistory` (line 1), then using this object to store and merge the results of method invocations (lines 3 and 7) and checking for exceptions when appropriate (lines 4 and 8). Apart from this exception control, the only differences between the code for managed classes and the code for Arjuna user classes are firstly the substitution of the Arjuna's `setlock` statement (line 2, Figure 8.2) by the object manager call `make_volatile` (line 3, Figure 8.3), and secondly the addition of the object manager call `make_permanent` (line 7, Figure 8.3).

The object manager call `make_volatile` basically fetches the object state (using the plex manager) from the node where the corresponding plex resides, sets a lock (*write* lock in the example) on the plex, and causes the object state to be locally unpacked to enable its manipulation by the managed class' method. The object manager call `make_permanent` packs the object state and sends it (using the plex manager) to the corresponding plex to write it as the current object state of the mobile object. When the atomic action *A* is ended (line 8) or aborted (line 9) the lock on the plex is released and all modifications (in the active and passive parts of the mobile object) are committed or ignored, as appropriate. If the atomic action *A* is aborted in line 9 then everything done by the statements between lines 3 and 7 inclusive, is undone. That means the object state is restored to the state it had previous to the object manager call `make_volatile` in line 3. (This recovery mechanism is implemented by simply making the active part of a mobile object an Arjuna recoverable object, i.e., the class `Object` is subclass of the class `LockManager`.)

Obviously, this scenario considers only a trivial situation: the atomic action *A* is the outermost one, i.e., *A* is not nested within any other atomic action,

```
01      AtomicAction A; A.Begin();
02      if (setlock(new Lock(WRITE)) == GRANTED)
03          { // Modify object attributes
04              ...
05              A.End();
06          }
07      else A.Abort();
```

Figure 8.2 Typical structure of Arjuna user classes' methods

```
01      OpHistory* oph = new OpHistory; // Exception handling
02      AtomicAction A; A.Begin();
03      *oph += make_volatile(WRITE); // Object manager call
04      if (oph->normal())
05          { // Modify object attributes
06              ...
07              *oph += make_permanent(); // Object manager call
08              if (oph->normal()) A.End();
09              else A.Abort();
10          }
11      else A.Abort();
```

Figure 8.3 Typical structure of managed classes' methods

and also there is no nesting of object manager calls. If A was nested into an atomic action B then the lock on the plex should be held until the end of B , and the commit of A should not cause permanent effects until B has ended as well. This control of nested atomic actions is realised by the Arjuna's atomic action mechanism, whereas the control of nested object manager calls is realised by the object manager itself. For example, let us suppose that the method shown in Figure 8.3 invokes, in line 6, another method of the same object and with the same structure. In this case, firstly the object manager call `make_volatile` in the inner invoked method cannot fetch the object state from the plex as the object state in the active part is already under modification, and secondly the object manager call `make_permanent` should not update the plex as the caller method can still modify the object state.

In fact, the control of nested object manager calls can be very complex, depending on the combination of situations, such as when the mobile object is being created, when the mobile object is being retrieved, when a method writes on the object state or simply reads it. All these situations are captured by the transition diagram for object state in the active part of mobile objects shown in Figure 8.4. The initial states **CREATION** and **RETRIEVAL** correspond to the situations when the mobile object is being created or retrieved, respectively. The state **MODIFIED** corresponds to the situation when the object state in the active part is "ahead" of the object state in the passive part (the plex) of the mobile object. The state **NORMAL** corresponds to the situation when the object state in the active part is identical to the object state in the passive part of the mobile object, and this must be the final state.

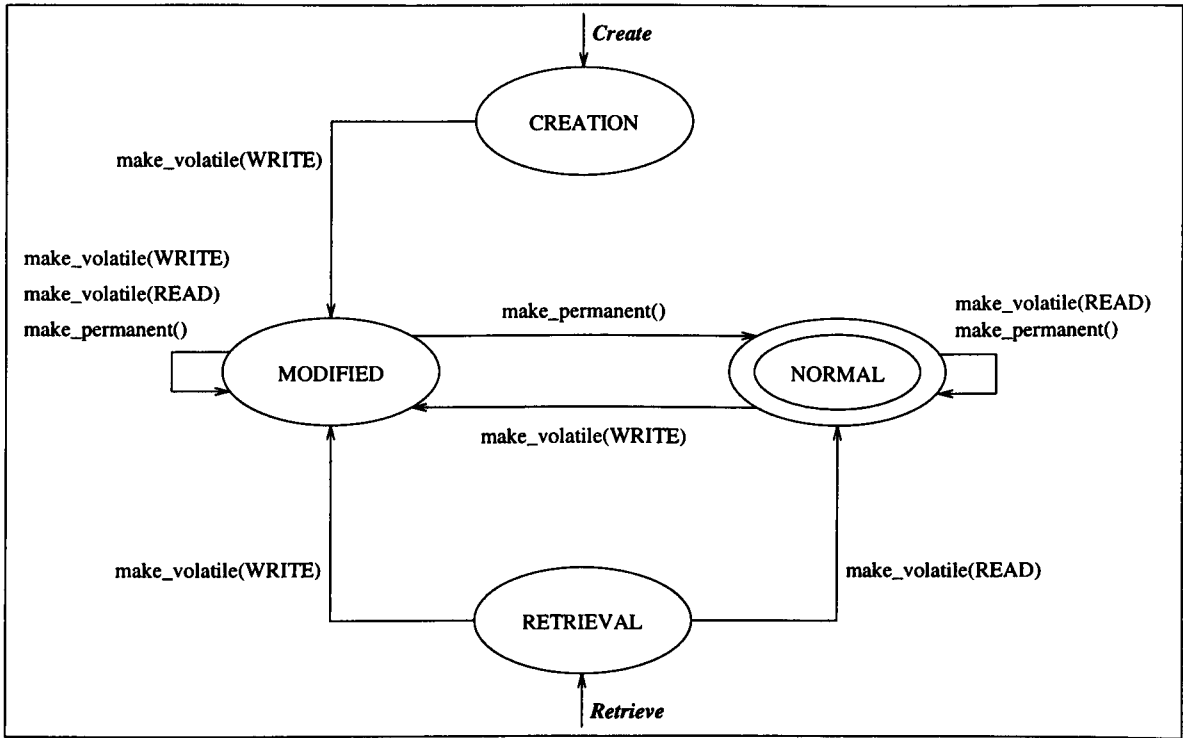


Figure 8.4 Transition diagram for object state

The transition diagram is relatively simple because of an auxiliary structure that permits resolution of certain situations of ambiguity. For example, if the method *makePermanent* is called when the state is **MODIFIED** what should be the next state? Using only the information in the transition diagram it could be either **NORMAL** or **MODIFIED** itself. The decision, in this case, depends on the circumstances of the invocation of the corresponding *make_volatile*. If the *make_volatile* was invoked when the state was already **MODIFIED** then there is no change of state, otherwise the next state is **NORMAL**. Therefore, the object manager decides state transitions with the help of a stack representing the history of all object manager calls. If the managed class code has its all calls properly balanced, then not only is the final state **NORMAL** but the stack is also empty.

8.1.2 Views and Indices

As discussed in Chapter 7, object engines maintain two types of indices: relationship indices and attribute indices. Accordingly, the implementation of relationship indices is realised by a class named `RelationshipIndex`. In the case of attribute indices, for practical reasons, a distinction is necessary according to the primary type of the attribute. Currently, the types supported are string and integer, and the corresponding classes `StringAttributeIndex` and `IntegerAttributeIndex`, respectively. Views, on the other hand, are simply implemented by a class named `View`. The structure of views and indices as well as their relations are illustrated using generic examples in Figure 8.5, for attribute indices, and Figure 8.6, for relationship indices.

In Figure 8.5, a class named `C` with attributes `s` and `i` of types string and integer, respectively, has three instances named `c1`, `c2` and `c3` (representing three UUIDs). Also, there are three meta-objects that map the class `C`, including its attributes. The meta-objects that map the attributes contain references to the respective indices.

In Figure 8.6, two classes named `A` and `B` are related to each other with roles `RA` and `RB`, respectively. There are three instances of each class which are related to each other as indicated by the double-headed arrows shown in dotted lines. The relationship between `A` and `B` is mapped by three meta-objects. The meta-object of class `Relationship` has references to relationship indices, one for each direction of the relationship.

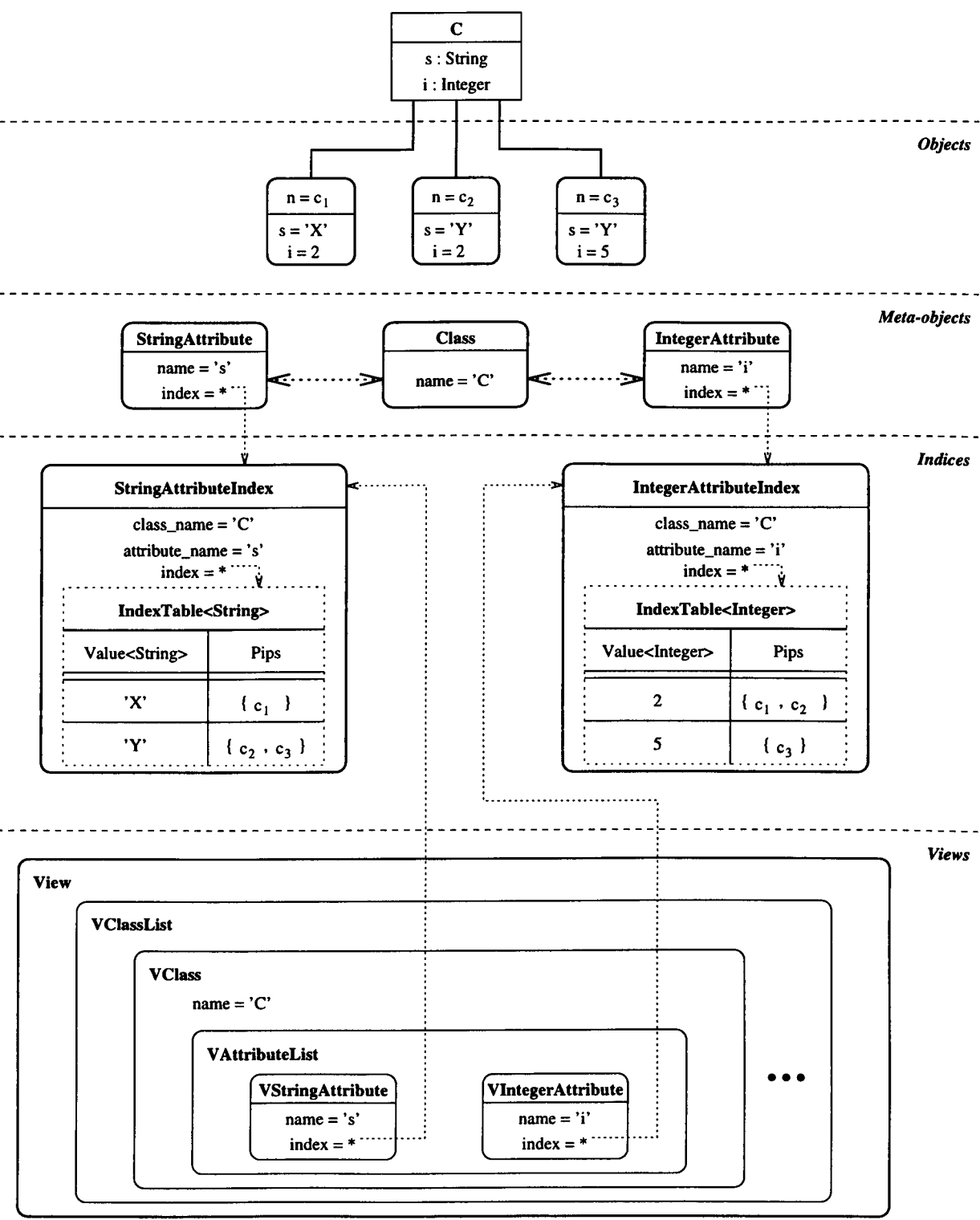


Figure 8.5 Attribute indexing implementation – a sample

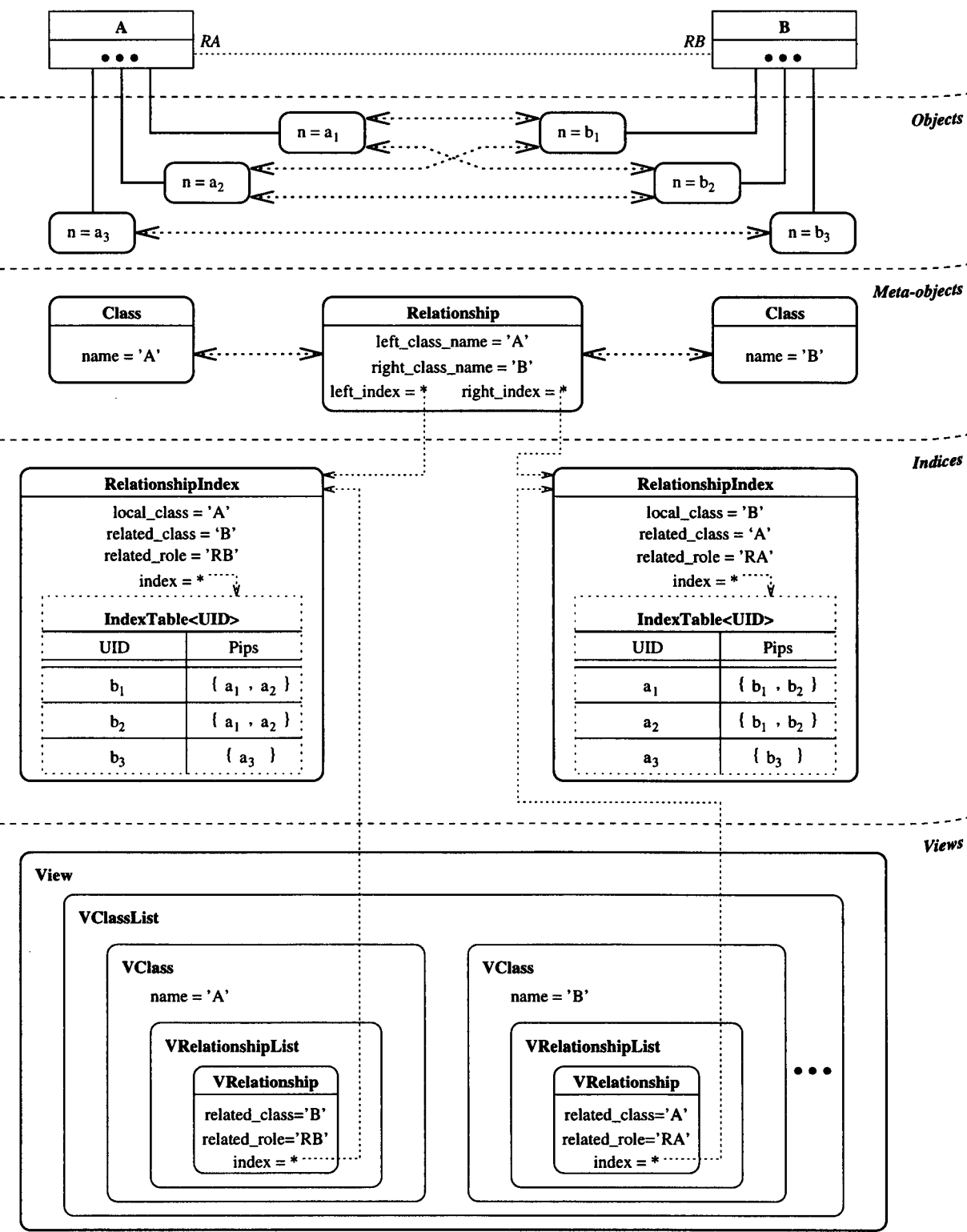


Figure 8.6 Relationship indexing implementation – a sample

In both figures, we can note the presence of a structure called **Pips** on the right-hand side of tables contained in indices. This structure is a set of special object references. Such a special reference, called *pip*, consists of the UID of the object, the name of its most specific class and the name of its home node. This information provided by a pip is sufficient to retrieve the object without the need to interact with the name server. (The name server is only consulted when the information contained in the pip is stale. In this case, the correct information provided by the name server is used by the plex manager and the pip is updated.) For simplicity, we do not use specific notation to indicate whether a UID, such as c_1 , really denotes a UID or a pip, as the context in which they appear is sufficient for the understanding.

View

An instance of **View** contains a list of classes. Each class has a name and contains a list of attributes (only attributes designated as keys in the schema) and a list of relationships (only relationships designated as keys in the schema). (The list of attributes and the list of relationships of a view are shown separately in Figure 8.5 and Figure 8.6, respectively). Each attribute has a name and a reference to the corresponding index. Each relationship has the name of the related class, the name of the related role and a reference to the corresponding index.

StringAttributeIndex

An instance of **StringAttributeIndex** contains a table (instantiated from a C++ template with parameter **String**) of string values mapped to sets of pips; a value is mapped to a pip if the object referenced by the pip has the corresponding attribute with that value. For example, in Figure 8.5, the value **X** is mapped to c_1 because the object named c_1 has the attribute **s** with value **X**.

IntegerAttributeIndex

An instance of **IntegerAttributeIndex** is similar to an instance of **StringAttributeIndex**, except that its table maps integer values, rather than string values.

RelationshipIndex

An instance of **RelationshipIndex** has the names of the local class, the related class, the related role and a table (instantiated from a C++ template with parameter **UID**) that maps **UIDs** of instances of the related class to pips of instances of the local class; the **UID** of an instance of a related class is mapped to the pip of an instance of a local class if the instances are related with respect to the class relationship. For example, the **UID** b_1 is mapped to the pips a_1 and a_2 as the object named b_1 is related to the objects named a_1 and a_2 .

8.2 User Object Manipulation

The manipulation of user objects by client programs requires mechanisms for proper management of object state changes, such as updating indexing information when there is assignment of new values to attributes which are indexed, and also for the provision of an easy-to-use interface, for example creating object relationships according to the established bi-directional semantics and, more importantly, for querying objects. For these reasons, every user object has a built-in set of methods which are made available to client programs through inheritance, i.e., the classes of user objects inherit these methods from a standard class and, therefore, client programs can invoke them. Since user objects are already subclasses of the class `Object` (they are mobile objects), we simply provide the methods for object manipulation by augmenting the object manager.

Let us introduce the methods for object manipulation through an example: the schema for bibliographical references shown in Figure 4.13. For simplicity, the C++ code presented in this Section is presented in a simplified form – exception handling is not included – for readability. Firstly, we recall that a client program must initiate interaction with an object engine by retrieving a context and then a view. Let us suppose that a client program wants to work in the context named “ComputingDepartment” and then manipulate the view named “BibliographicalReferences” which, obviously, corresponds to the schema for bibliographical references. The C++ code for this initialisation can be written as follows.

```
Context context ("ComputingDepartment", RETRIEVE);
```

```
View* view = context.get_view("BibliographicalReferences");
```

Henceforth, the variable `view` can be provided as a parameter when instantiating objects, thereby permitting the object to interact with the view for the purposes of indexing information management and query resolution.

Object Creation

The object manager provides a constructor that permits an object to be created by specifying as a parameter an expression containing the name of the class followed by a list of attribute-value pairs. Such an expression is called *assignment expression*. The following C++ code creates an object of class `Book` having the attribute `title` with value "Object-Oriented Software Construction", attribute `year` with value 1988 and the attribute `publisher` with value "Prentice Hall".

```
Book b ("Book(title = 'Object-Oriented Software Construction' &&  
        year = 1987 && publisher = 'Prentice Hall')", view, CREATE);
```

Henceforth, the variable `b` refers to the object of class `Book` and can be used in the client program to manipulate the object. The corresponding indexing information is automatically inserted by invoking appropriate methods of the view given as a parameter. This view is also "remembered" by the object manager for possible future modifications.

Attribute Update

The object manager provides a method named `put` that accepts as a parameter an assignment expression. The following C++ code updates the attribute `year`

of the object of class **Book** created above to 1988. The corresponding indexing information is automatically updated by invoking methods of the view provided when the object was instantiated.

```
b.put("Book(year = 1988)");
```

Attribute Access

The object manager does not provide methods to get the value of object attributes. This facility is provided by the code automatically generated (using meta-objects) for classes. For each attribute of a class there is a corresponding method whose name is formed by the prefix **get_** and the name of the attribute. These methods are usually called *accessors*. The following C++ code gets the values of the attributes defined for the object of class **Book** above.

```
String title      = b.get_title();  
int    year       = b.get_year();  
String publisher  = b.get_publisher();
```

Relationship Creation

The object manager provides a method named **relate** to create a relationship between two objects, with automatic update of the corresponding indexing information. The method must be applied to one of the objects and the parameters must include the other object and the role of the other object in the relationship. Let us suppose that the client program has a variable *i* that refers to the object of class **Individual** that corresponds to the author of the book represented by the

object denoted by the variable *b* above. The following C++ code relates both objects accordingly.

```
b.relate("Author", i);
```

An equivalent way of creating this relationship is given as follows.

```
i.relate("BookAsAuthor", b);
```

Relationship Deletion

The object manager provides a method named **unrelate** which has opposite effect of the method named **relate**. The following C++ code deletes the relationship between author and book created above.

```
b.unrelate("Author", i);
```

Relationship Traversal

The object manager provides a method named **get_related_pips** which takes as parameters a related role and a class name, and returns a set of pips that refer to related objects which are instances of the specified class. Then, the set of pips can be used for retrieving the related objects using another constructor provided by the object manager. The following C++ code gets a set of pips that refer to the authors of the book denoted by *b* which are instances of the class **Individual**, and assigns this set to the variable **authors**. Next, the first pip is extracted from the set and used for retrieving the referred object, i.e., an instance of **Individual** that is author of the book denoted by *b*. The client program can iterate over the set

and retrieve all authors by making use of methods `next` and `cardinality` provided by the class that implements sets.

```
Pips authors = b.get_related_pips("Author", "Individual")
Pip p = authors.first(); // extracts the first pip
Individual first_author(p, view);
```

Object Retrieval

The object manager provides a constructor that permits an object to be retrieved by specifying as a parameter a *query* expressed in the language described in Section 8.3. The query is resolved as explained in Section 8.4 and one of the pips from the resulting set is selected at random. Next the object state is fetched from the corresponding plex and locally unpacked for manipulation by the client program. The following C++ code retrieves an instance of `Book` whose title contains the word “software” and invokes its method `print`.

```
Book book ("Book(title % 'software')", view, RETRIEVE);
book.print();
```

If the client program wishes to manipulate all instances that satisfy a query rather than just one instance as described above, then the class `ObjectSet` must be used instead. The class `ObjectSet` accepts as a parameter a query expression and then instantiates all objects that satisfy the query. Thereafter, the client program can iterate over the set to manipulate the objects individually. The following C++ code retrieves all instances of class `Book` whose titles contain the word “software” in a set assigned to a variable `s`. Next the set `s` is iterated to invoke the method `print` for each object.

```
ObjectSet s ("Book(title % 'software')", view);
Book* b;
for (i = 0; i++; i < s.cardinality())
{
    b = (Book*) s.next();
    b->print();
}
```

8.3 Query Language

An object query, or simply query, is a declarative specification of objects according to their properties with the purpose of facilitating object retrieval. Therefore, queries are essential for object engines. We have defined a language to express queries against schemas modelled according to the technique presented in Chapter 4 and formalised in Appendix A. A query is formulated as a Boolean combination of predicates expressed in terms of classes, attributes and relationships. The result of a query is a set of pips that refers to objects whose properties are in conformity with the specified predicates. Conceptually, the result of a query is a set of objects. A query is *schema conservative*: neither objects nor classes are created as a result of a query. Thus, a set of objects obtained as a query result is composed of objects which are existing instances of existing classes. We designed the query language in a fashion that resembles the expressions of the C++ programming language. Actually, all syntactic constructs of the query language are found in C++. For example, the following query retrieves all objects which are instances of class `Individual` and have attribute `surname` equal to "Meyer".

Individual(surname == 'Meyer')

As another example, the following query retrieves all objects which are instances of the class **Book** and have the attribute **year** greater than 1980.

Book(year > 1980)

Below we illustrate the basic constructs of the query language with some queries formulated against the schema for bibliographical references shown in Figure 4.13. These basic constructs can be combined to formulate more complex queries, as defined by the grammar presented in Appendix F.

Class Expressions

A class expression is the mandatory construct in any query. It consists of the name of a class, the *target class* of the query, followed by a an expression surrounded by parentheses containing predicates about the target class. For example, in the query

Book(title % 'software' && year > 1980)

the target class is **Book** and the expression `title % 'software' && year > 1980` contains predicates about the class **Book**, i.e., the query specifies the objects which are instances of the class **Book**, have the word “software” as substring of its title and have been published after the year 1980.

Attribute Predicates

An attribute predicate is a triplet $\langle \text{attribute} - \text{name operator value} \rangle$, such as `year > 1980`. The value must conform with the type of the attribute (defined in the schema). An attribute can be either an integer or a string. The operators for attributes are the standard relational operators `==, !=, <, >, <=, >=`. In addition, for string attributes the operator `%` which means substring is supported.

Boolean Combination of Predicates

The Boolean operators *and* (`&&`) and *or* (`||`) can be used to combine predicates, thereby forming larger predicates which can be, recursively, combined into new predicates. For example, the following is a valid combination of predicates:

```
((title = 'Through the Looking Glass' && year > 1870) || (title % 'Alice'))
```

Superclass Access

The target class of a query can be a class that has subclasses. In this case, the objects retrieved can have as their most specific class any of the subclasses of the target class. For example, the following query retrieves all instances of class `Reference` whose titles contain the substring “object”. Since the `Reference` is superclass of the classes `Book`, `Journal` and `Article` then the result of the query may contain objects of any of these classes.

```
Reference(title % 'object')
```

Attribute Cast

In the cases where the target class is a superclass it may be convenient to be more specific about the attributes of the superclasses which are of interest. For example, the following query retrieves all instances of the class **Reference** which are either publications published in 1988 or articles whose page numbers are 16 to 33.

Reference([Publication]year = 1988 || [Article]pages = '16-33')

Associative Access or Nested Query

Relationships are useful for specifying *associative queries*, i.e., objects can be retrieved according to the relationships between classes. Basically, instances of a class α are designated through attributes and relationships of instances of a class β that is related to α , such that the instances of α and β are related. For example, the following query retrieves all instances of class **Book** whose authors have surname *Meyer*.

Book(Author(surname = 'Meyer'))

This query can be seen as a composition of nested queries. The outermost query has **Book** as the target class, while the innermost query has **Individual** as the target class. Although this example has just one level of nesting, in general there is no restriction to the level of nesting in queries. This permits the formulation of queries with very complex paths through a class hierarchy. We should note that nested queries are more general than the traditional *path expressions* supported by query languages in database systems. In fact, the query language also supports

a syntax for associative queries that is similar to path expressions. The example above could also be written as follows.

```
Book(Author::surname = 'Meyer')
```

However, the following nested query cannot be written using the path operator “::” without defining several path expressions.

```
Book(title % 'software' || Author(surname = 'Meyer' && forenames = 'Bertrand'))
```

Role Cast

In associative queries it may be convenient to be more specific about the related class, i.e., it may be interesting to navigate through a subclass of a related class. For example, the following query retrieves instances of the class **Library** that contain books whose titles contain the substring *modeling*. We should note that the relationship in the schema is specified between the classes **Library** and **Publication**, which is superclass of **Book**.

```
Library( [Book]Publication(title % 'modeling'))
```

Set Operations

A query can be formulated with more than one target class. In this case, all class expressions specified must be combined with set operators *union* (`()`) and *intersection* (`&`). Similar to the combination of Boolean operators, the combination of set operators can be used form larger queries. For example, the following query retrieves all instances of **Article** whose titles contain the substring “spring” and all instances of **Book** published later than 1970.

```
Article(title % 'spring') | Book(year > 1970)
```

Operator Associativity and Precedence

The associativity of all operators is left to right. The precedence of operators is summarised in Table 8.3. (The path operator has the higher precedence.)

Function	Operator
path operator	::
relational operators	==, !=, <, >, <=, >=, %
logical AND	&&
logical OR	
set intersection	&
set union	

Table 8.1 Query language operators precedence

8.4 Query Resolution

A query is resolved by creating a corresponding tree representation and then reducing the tree. Each reduced subtree is replaced by the corresponding resulting set of pips until eventually the set of pips corresponding to the whole query is obtained. All nodes of the tree, except the leaves, are instances of the classes depicted in Figure 8.7. For example, let us consider the following query:

```
Article(title % 'pollution' && Author(surname == 'Green'))
```

This query is translated into the tree depicted in Figure 8.8. For this discussion, we labelled the nodes as indicated. The node 1, an instance of the class **ClassOp**, corresponds to the target class of the query, the class **Article**. The remaining nodes are arranged according to the Boolean combination of predicates and the precedence of operators. The leaves of the tree are arranged as attribute-value pairs; each pair corresponds to the children of a relational operator. The leaves 4 and 5 form the attribute-value pair of the relational operator “%” on node 3, while the leaves 8 and 9 form the attribute-value pair of the relational operator “==” on node 7. The subtree rooted at the instance of the class **RelationshipOp**, the node 6, corresponds to the nested query. Thus, the relational operator under the node 6, i.e., the operator “==” on node 7, must be resolved with respect to the innermost target class, i.e., the attribute **surname** must be interpreted as an attribute of the class that is related to the class **Article** and has role **Author**, rather than an attribute of the class **Article**. If we check the schema we will learn that the target class of the innermost query is **Individual**. Although this information is not explicitly represented in the tree, there is enough information for the view to find this out, since the view has a representation of the target schema.

The query is resolved by in-order traversal of the tree and reduction of each subtree. The resolution starts at the node 1 which simply passes the current target class name (**Article**) to its child, the node 2, and awaits the result. The node 2 is an *and* operation, so it has to make the intersection of the sets of pips corresponding to each of its children, the nodes 3 and 6. Thus, the node 2 sequentially forwards the current target class name (**Article**) to its children in order to obtain the two

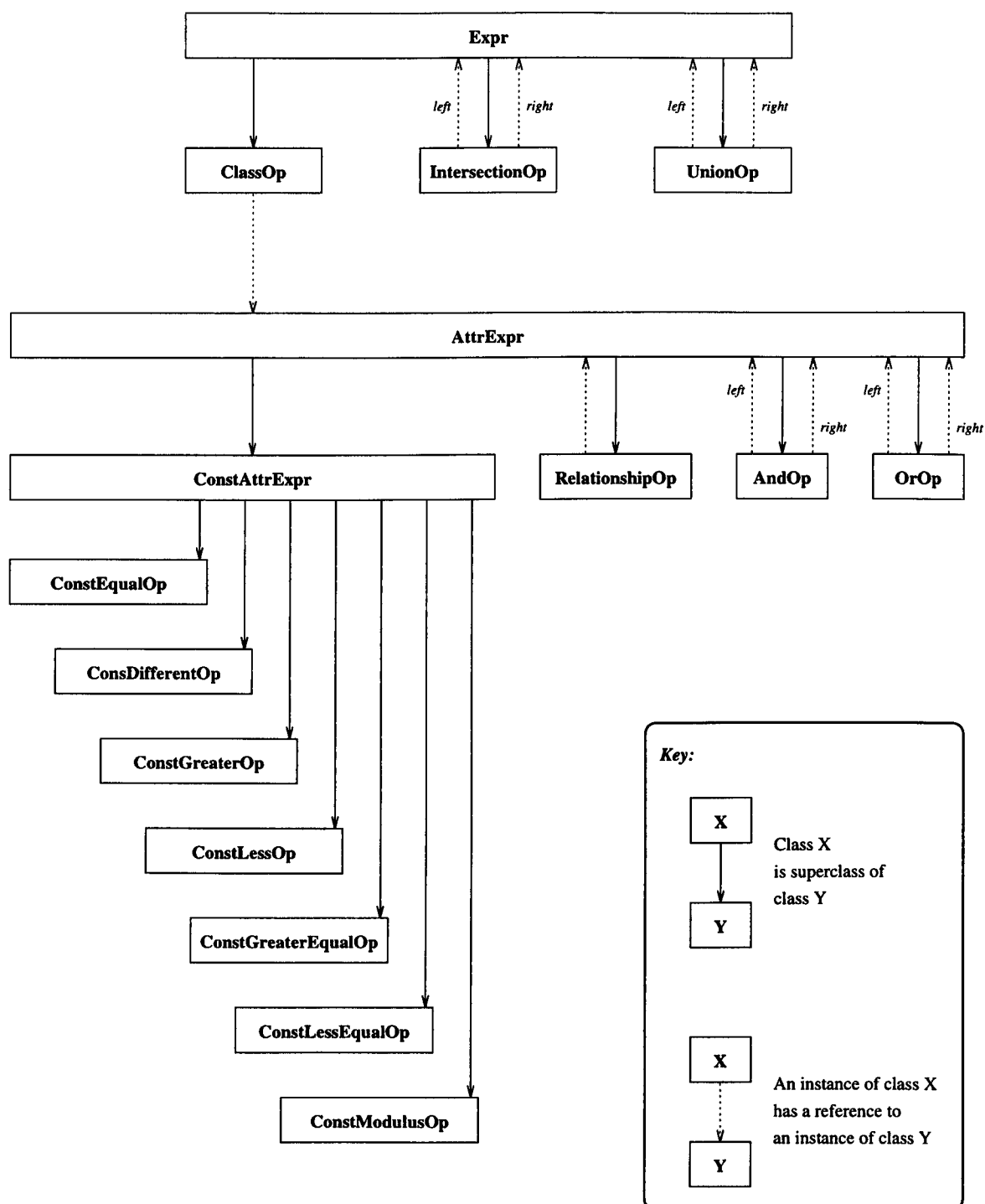


Figure 8.7 Class hierarchy for the tree representation of query expressions

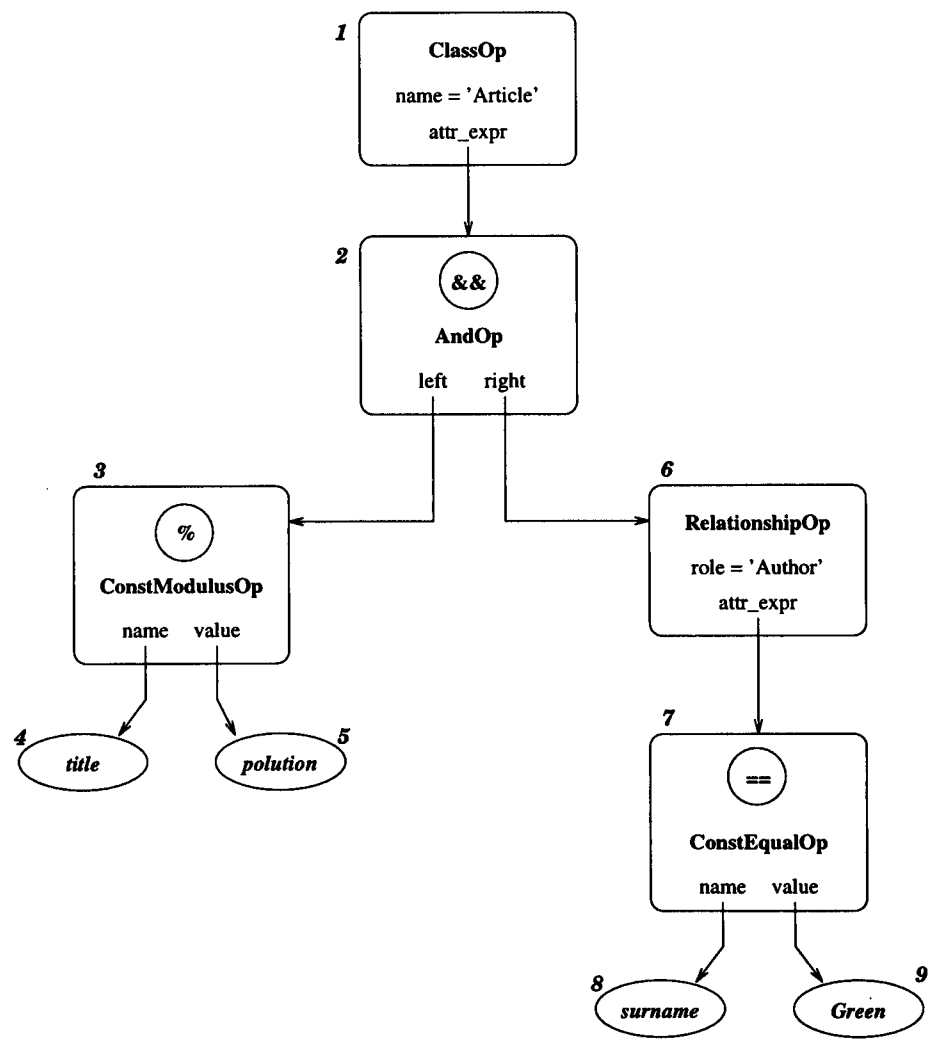


Figure 8.8 A tree representation of a query

sets. Firstly, the node 2 forwards the current target class to the child on its left side, the node 3, and awaits the corresponding set of pips. The node 3 is the relational operator “%” and, therefore, can be reduced. The node 3 takes the attribute-value pair represented by its children (nodes 4 and 5) and the name of its target class (Article) to form the predicate $\langle \text{Article}, \text{title}, \%, \text{pollution} \rangle$ which is then submitted to the view. The view takes the predicate and searches in its internal structure (see Figure 8.5) for the attribute title of class **Article**. If the

attribute is found, then the view submits a request to the corresponding attribute index giving as parameter the operator “%” and the string “pollution”. The index returns a set of pips which refer to all instances of **Article** whose attribute **title** contains “pollution” as substring. This resulting set of pips is then returned by the view to the node 3, completing its reduction, thereby providing the left set of pips for the node 2. Then, the node 2 forwards the current target class (**Article**) to the child on its right side, the node 6, and awaits the corresponding set of pips. The node 6, which is the root of a nested query, takes the received target class (**Article**) and the related role (**Author**) to ask the view for the new current target class, obtaining as response the class name **Individual**. Thus, the node 6 forwards this new current target class to its child, the node 7, and awaits the corresponding set of pips. The node 7, which is a relational operator “==”, is reduced in a similar fashion as the node 3 was reduced. The result of it is the set of pips that refer to instances of **Individual** whose attribute **surname** has value equal to “Green”. Now the node 6 takes this set of pips that refers to instances of **Individual**, the name of the related class (**Article**) and its local role (**Author**), and finally asks the view to return the set of pips that refers to instances of **Article** which are related to the instances of **Individual** referred by the obtained set of pips. The view searches in its internal structure (see Figure 8.6) for the relationship of the class **Article** that has **Author** as the related role. If the relationship is found then the corresponding relationship index is used for obtaining the set of pips that refer to instances of **Article**, as requested. This set is then returned to node 6, completing its reduction, thereby defining the right set of pips of node 2. Now, the node 2 makes the intersection of both sets of pips, completing its reduction. Finally, the node 1 takes this set of pips and the whole tree is reduced.

8.5 Object Engine Set-up

In Chapter 6, we presented the implicit recursion of the meta-model as an interesting bootstrap problem. To reiterate, because every class has to be mapped to meta-objects for it to exist, the existence of meta-classes is required in order to enable the creation of meta-objects. However, meta-classes are classes which also need to be mapped for them to exist. This problem has been simply solved at the conceptual level by postulating the existence of meta-classes. Accordingly, *Stabilis* provides an implementation for the meta-classes, thereby permitting the creation of the set of meta-objects that map the meta-schema at the first stage of an object-engine set-up. At the programming interface, an object engine is simply set-up by creating the corresponding context. For example, the following C++ code creates a context named “ComputingDepartment”.

```
Context    c ("ComputingDepartment", CREATE);
```

The constructor of the class **Context** creates a set of meta-objects that maps the meta-schema, the corresponding indices and a meta-view (the view named “Meta”). Actually, indices are automatically created during the creation of meta-objects; instances of the classes **Attribute** and **Relationship** create the corresponding index when they are created. We should note that, since meta-objects are instances of meta-classes, meta-objects must be indexed by the indices that they create. This circular dependency has been discussed in Chapter 3, and illustrated in Figure 3.3. The solution to this problem is to postpone the indexing of the meta-objects, i.e., the constructor of the class **Context** invokes appropriate methods of instances of **Attribute** and **Relationship** to fix the indices. Finally, we must recall that contexts, views and meta-objects are mobile objects. For this reason,

the creation of a context requires the appropriate infrastructure for object mobility, i.e., there must exist a physical installation of plex manager, including a name server.

8.6 Application Development

The development of an application encompasses the following steps:

1. devise a schema
2. create the meta-objects that map the schema (indices automatically created)
3. generate a view corresponding to the schema
4. generate code for classes of the schema (this can be done automatically)
5. if appropriate, generate interactive query interpreter (this can be done automatically)
6. create programs to manipulate the database designated by the schema (the programs should use the code for classes)

We will explain each of these steps through an example application that we developed as the demonstrator or proof of concept. The application domain that we have chosen is bibliographical information based on the types of entries for bibliography citation defined for `BIBTEX` [35]. An example of a `BIBTEX` file (which is a network resource) is shown in Figure 8.9. The schema that we devised to model bibliographical references is shown in Figure 8.10. The schema, named `Dbib`, contains 24 classes, with class hierarchies of a depth up to 5, and several associations and aggregations.

```

@Book{   Meyer88,
        title = "Object-Oriented Software Construction",
        author = "Bertrand Meyer",
        publisher = "Prentice-Hall", year = 1988}

@Book{   Rumbaugh91,
        title = "Object-Oriented Modeling and Design",
        author = "Rumbaugh, J. and Blaha, M. and Premerlani, W. and Eddy, F. and Lorensen, W.",
        publisher = "Prentice-Hall", year = 1991}

@Article{ Meyer86,
        title = "Genericity versus Inheritance",
        author = "Meyer, Bertrand",
        journal = "ACM Sigplan Notices", year = 1986, month = October,
        pages = "391--405"}

@Article{ Koenig90,
        title = "Exception handling in C++",
        author = "Andrew Koenig and Bjarne Stroustrup",
        journal = "Journal of Object-Oriented Programming", year = 1990, month = July,
        pages = "16--33"}

```

Figure 8.9 Example of a network resource: a $\text{BIB}\text{T}_{\text{E}}\text{X}$ file

The schema `Dbib` is mapped to 104 meta-objects, and it needs 13 indices. The following C++ code illustrates how these meta-objects can be created by an administrator program. (The corresponding indices are automatically created.) The names chosen for the variables should make the code self-explanatory.

```

Context context("ComputingDepartment", RETRIEVE);

View* m = context.get_view("Meta");

Schema s_Dbib("Schema(name == 'Reference')", m, CREATE);

Class c_Book("Class(name = 'Book')", m, CREATE);

StringAttribute a_Book_publisher

    ("StringAttribute(name = 'publisher' && key = 0)", m, CREATE);

a_Book.relate("Attribute", a_Book_publisher);

s_Reference.relate("NonRootClass", c_Book);

```

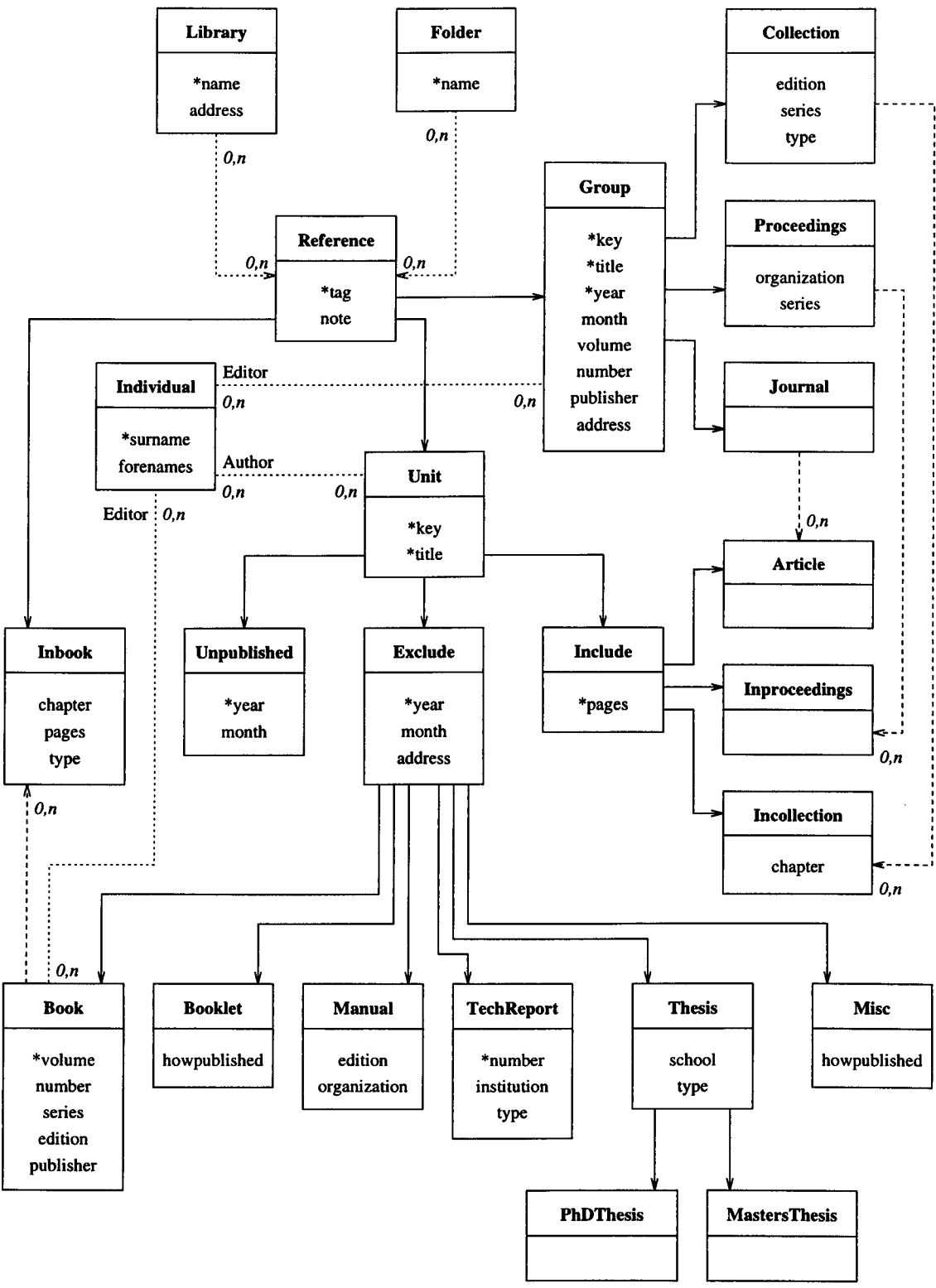


Figure 8.10 Schema for bibliographical references

Alternatively, these meta-objects can be created using the query interpreter for the meta-view which is called `parla`. The use of `parla` is illustrated below. We can observe that the commands to query interpreters are very similar to C++ statements. Also we can observe that the view used for creating objects is implicitly defined.

```
parla> Schema s_Dbib("Schema(name == 'Reference')", CREATE)
parla> Class c_Book("Class(name = 'Book')", CREATE)
parla> StringAttribute a_Book_publisher
      ("StringAttribute(name = 'publisher' && key = 0)", CREATE)
parla> a_Book.relate("Attribute", a_Book_publisher)
parla> s_Reference.relate("NonRootClass", c_Book)
```

Once the meta-objects have been created, views can be generated for the schema `Dbib` and any other self-contained schema defined by the meta-objects. The following C++ code generates a view for the schema `Dbib` and registers it with the appropriate context. The variable `pip_view_Dbib` is a reference to the view obtained when the view is created and passed to the context for the registration.

```
Pip pip_view_Dbib;
View v_Dbib(s_Dbib, pip_view_Dbib, CREATE);
context.enter_view("Dbib", pip_view_Dbib);
```

The meta-objects also permit the generation of C++ code for the corresponding classes. The code generated for a class includes the declaration part, the implementation of constructors to interact with the object manager, the accessors (methods for reading attribute values) and special methods for packing/unpacking

the object state, as required by the Arjuna system. Code can be generated separately for each class or for all classes of a given schema, as appropriate. The following C++ statement generates code for all classes of the schema *Dbib*. The parameter specified in the method invocation defines the directory of the file system where the code will be created.

```
s_Dbib.gen_code("/usr/home/n04ao");
```

The code generated for classes can be used for the construction of programs to manipulate the database of bibliographical references. A program that can be automatically constructed is a query interpreter specific for each schema. This query interpreter is constructed simply by linking the code for classes with a "skeleton" provided by *Stabilis*. (An example of this query interpreter, named *parla*, has been shown above for manipulating meta-objects. A query interpreter for *Dbib* would have the same functions of *parla*, except that the database manipulated would be distinct.) Collector programs should also benefit from the generated code. For example, we created a tool that generates programs to create/update objects corresponding to bibliographical information extracted from *BIBTEX* files. Also, we created a graphical interface to a partial view of *Dbib*. The main "window" of the graphical interface is illustrated in Figure 8.11.

8.7 Performance

Some performance figures of an object engine for bibliographical references containing approximately 1,000 objects are shown in Table 8.2. The object engine runs distributed over a set of workstations connected by an 10 megabits/s-

Ethernet LAN. The query times include the following: parse the query, invoke remote operations on indices, search the indices, send partial results back to client node, and merge the partial results. The retrieve times are average times for moving objects from plexes in remote nodes to client programs. These retrieve times vary according to object size (which depends on attributes and relationships) — their average size is approximately 1 Kbyte.

Query expression	Query time (ms)	Retrieve time (ms)
<i>Article(title %'object')</i>	216	67
<i>Journal(title %'comput')</i>	125	120
<i>Journal(year > 1980)</i>	107	120
<i>Journal(title %'comput' && year > 1980)</i>	181	128
<i>Book(Editor(surname %'e'))</i>	188	141

Table 8.2 Performance of the object engine for bibliographical references

8.8 Conclusions

Stabilis fully implements the architecture for object engines. The programming interface is simple, integrated within a standard programming language (C++), and provides good distribution transparency, thereby making it easy to write (collector, administrator and client) programs. The query language is seamlessly incorporated into C++; there is no impedance mismatch between data manipulation language and data computation language. Since, C++ is the only programming

language used in the implementation and also is the language at the programming interface, neither language extensions nor special compilers are necessary, thereby contributing to systems portability. Meta-objects and associated tools for automatic code generation permit fast program development. Stabilis has been used to implement an object engine for bibliographical references in order to validate the described architecture.

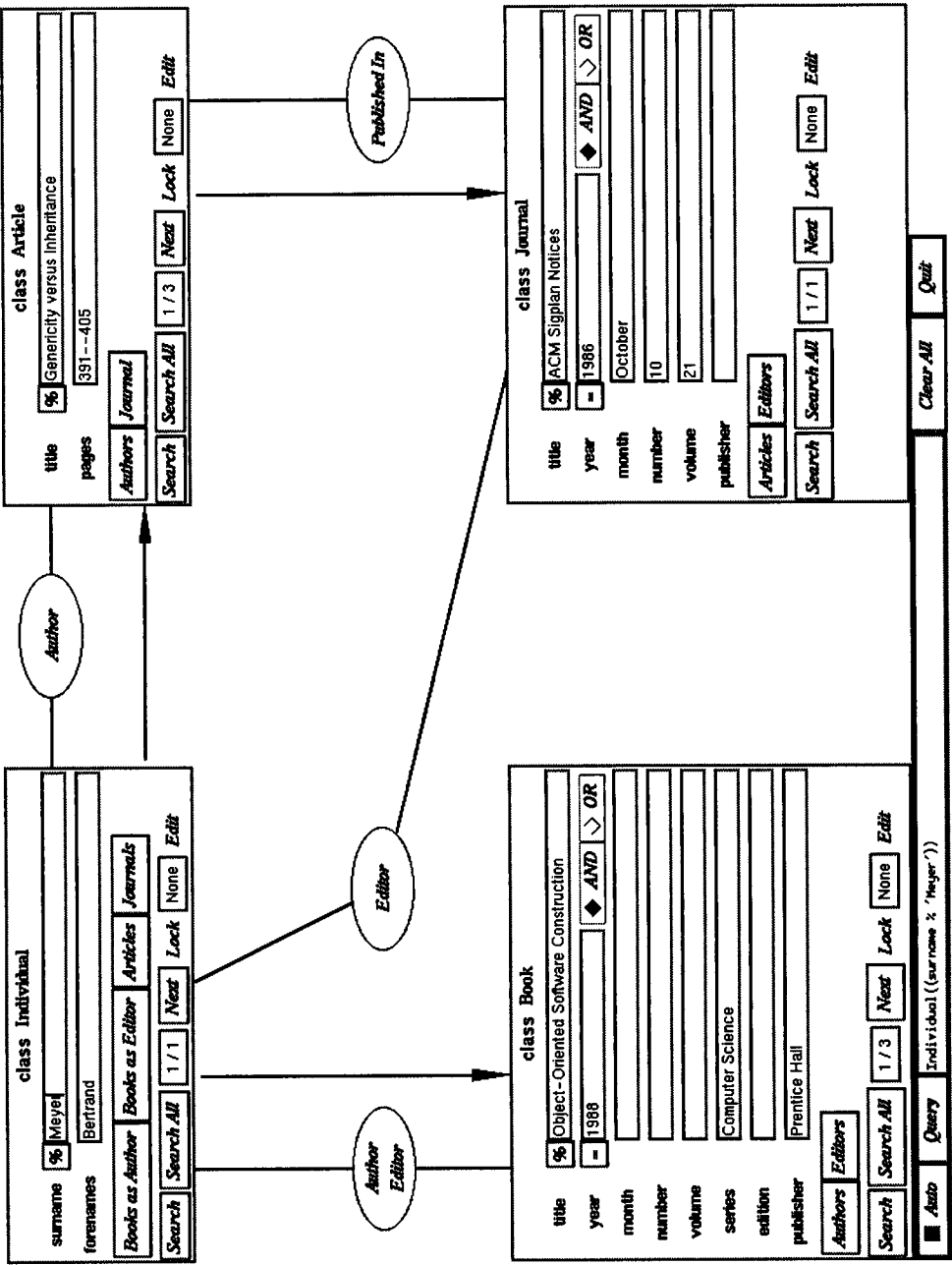


Figure 8.11 Graphical interface for bibliographical references

CHAPTER 9

Conclusions

Our thesis has concentrated on the problem of manipulating structured information contained in network resources which are located over large-scale distributed environments. In this Chapter we summarise the research which has been done, point out the main contributions of the research, give a brief account of the evolution of our ideas and experiments, and finally we suggest directions for future research and further development of the prototype system.

9.1 Thesis Summary

In Chapter 2 we discussed a number of different approaches to the manipulation of information in distributed systems. In Chapter 3 we introduced a novel category of meta-information systems called *object engines* in which structured information contained in network resources can be manipulated through an object-oriented interface, and with high availability and distribution transparency. In Chapter 4 we defined a simple set of concepts commonly accepted in object-oriented systems, and introduced a corresponding graphic notation to represent schemas that model information contained in network resources. In Chapter 5 we introduced a model

for flexible organisation of the class space in schema hierarchies, providing the basis for a formal definition of databases. In Chapter 6 we described a special schema called *meta-schema* which models information about classes and schemas, and whose instances, i.e., meta-objects, represent meta-data. In Chapter 7 we defined indices, views and contexts, to complete the description of object engine components. In Chapter 8 we described the implementation of a platform for constructing object engines, including the definition and use of a query language for object manipulation, the management of distribution, and a demonstration application.

9.2 Thesis Contributions

1. *Identification of critical issues in manipulating structured information.*

Typically, each of the approaches to the manipulation of information in distributed systems, discussed in Chapter 2, gives more emphasis to a particular issue, such as locating relevant network resources by making use of information semantics, providing high availability of services and providing transparency to distribution. Our investigation of such approaches has identified critical issues in constructing systems for manipulating structured information in large-scale distributed systems, especially in constructing systems according to the object-oriented paradigm.

2. *Consolidation of concepts found in several areas.*

The architecture we have defined for object engines consolidates concepts found in information discovery tools, distributed systems and object-oriented

databases, in a homogeneous object-oriented framework, i.e., all object engine components are described, implemented and manipulated as objects. This approach greatly simplifies the implementation, administration and use of object engines, for all components benefit from the high availability and consistency provided by the underlying distributed transaction facility, and all programs interact with object engines through a single interface.

3. *Efficiency, effectiveness and reliability of information systems.*

The use of object engines to locate structured information permits accurate query formulation, which contributes to increase the rate of relevant hits and to reduce the number of iteration steps to locate information, thereby preventing the user from information overload as well as saving in processing and communication costs. The homogeneous storage of information objects and meta-data (due to the reflexive architecture of object engines) makes it easy for users to navigate through meta-data to learn what information is available and how to formulate good queries. The combination of schemas, views and contexts for organising the information space provides a suitable framework for efficient and effective use of information in large-scale distributed systems. Object engines increase reliability due to the use of replication and transactional access to objects.

4. *Implementation of a platform for constructing object engines.*

The Stabilis toolkit described in Chapter 8 has been designed and implemented not only as *proof of concept* but also as a platform for real use. Our preliminary experiments with the toolkit indicate that Stabilis object engines are an effective means of manipulating information objects, due

both to the power of object-oriented modelling, and high availability and consistency provided by the underlying distributed transaction facility. The toolkit is highly portable and has an easy-to-learn interface since it has been implemented using only *standard* programming languages and operating systems. The implementation of the toolkit enhances the Arjuna programming system with an object query facility, an object mobility scheme integrated with the transaction mechanism and provision for automatic generation of code to pack/unpack object states. Furthermore, the toolkit demonstrates the adequacy of the object and action model of computation as a framework for writing fault-tolerant distributed applications.

9.3 Evolution of Ideas and Experiments

Basically, our research has been carried out in three well defined phases, each of them composed of a period of study followed by a period of experimentation. In the first phase we developed a simple version of Stabilis [12], as a programming exercise using the Arjuna system. That version provided for the automatic generation of a “query interpreter” for a given schema. Such a query interpreter had the form of a *graphical interface* with functions for the manipulation of instances of the classes designated by the corresponding schema. Although those query interpreters provided satisfactory facilities to create, modify and relate objects, the search interface was limited to specifying a keyword that should be substring of an attribute defined as the *primary key* of each class. Moreover, there was no support for meta-data management, nor information space organisation.

In the second phase, having learned from the initial experiment, we developed, practically from scratch, a more sophisticated version of Stabilis that supported queries expressed as a Boolean combination of predicates, incorporated the notion of meta-objects, and had a more optimised caching scheme. The main purpose of that version was its use by a rule-based system for the management of distributed programs called *Vigil* [11]. However, the approach taken to index information management and query resolution in that version led the system to have an unacceptable performance. Also, organisation of information space was still restricted to the notion of contexts.

In the third phase, we completely restructured the indexing scheme, extended the query language and introduced the notion of views in the system. The index information was moved from meta-objects to specialised structures. The effects of this were a better system modularisation, permitting the use of appropriate data structures to implement indices, thereby simplifying and dramatically improving the performance of index information update and query resolution. The extensions to the query language included nested query, casting and approximate match, in order to better suit the kind of queries about information contained in network resources. The purpose of introducing views were twofold: a means of organising the information space and, for efficiency reasons, a simplification in the representation of meta-data used during object manipulation. While the implementation realised in the first and second phases were a joint work with another researcher, the implementation realised in the third phase was an individual work by the author of this thesis. The current version of Stabilis has approximately 40,000 lines of source code.

9.4 Future Work

The research and implementation that we have done open a spectrum of opportunities for further development, including improvements in the current implementation, advances in the current functionality, and new applications of object engines.

9.4.1 Implementation Improvements

The following list enumerates some implementation issues that remain to be tackled in the Stabilis toolkit.

1. Implementation of indices using more sophisticated data structures, such as B-trees, in order to properly support large object bases.
2. Implementation of index servers that can handle multiple indices, similarly to the object state server (which can handle multiple object states), in order to reduce the number of processes in the system and reduce the latency in index operations, thereby improving the overall systems performance.
3. Use of nested top-level atomic actions¹ [38] to control access to global resources, such as indices. This is important to avoid such resources remaining locked (and thereby become unavailable to other users) in a long-running transaction. The use of nested top-level atomic actions, however, may require “anti-actions” to compensate their effects. For example, let us suppose

¹A nested top-level atomic action is an independent action started within another atomic action.

that an object is created within an atomic action A , and that the corresponding index update is realised by a nested top-level atomic action B , that is, B is started within A . If, for some reason, the atomic action A aborts (and then the object was not actually created) then an anti-action must undo the effects of the atomic action B , i.e., remove the index information which has been inserted.

4. Decentralisation of indices. Currently, indices are single global entities. A more general approach should permit a logically single global index to be implemented as a collection of cooperating indices.
 5. Implementation of object removal. Of course, object removal requires proper algorithms to ensure referential integrity. This is particularly simple to implement in *Stabilis* since all object relationships are bi-directional. However, the underlying transaction facility currently does not support object removal.
 6. Optimisation of query resolution. The current version does not make any effort to save on index access, nor to solve parts of queries in parallel. The provision of these features normally requires formal models for the object data model and for the query model. Since we already have defined a formal model for the object data model (Appendix A), an important step towards query optimisation has already been done.
 7. Automatic generation of graphical user interfaces. The availability of meta-data already permits the automatic generation of command-line-based interactive query interpreters. This same meta-data could perfectly be used for the generation of more user-friendly interfaces.
-

8. A straightforward extension to our work would be to make Stabilis object engines accessible via the World Wide Web (WWW). The main task in the provision of this service would be the development of appropriate servers to interface between object engines and WWW applications.

9.4.2 Functional Advances

Database Issues

The query language can be extended in many ways. Firstly, it can support a larger collection of basic types for attributes (in addition to integer and string), and also user-defined *tuple types*, such as a type to represent date. Secondly, it can incorporate traditional information retrieval techniques, such as ranked queries and sophisticated support for approximate match. Thirdly, it can be extended to support method call: this would permit very complex queries, and the query language would become extensible. Fourthly, a support for explicit *range variables* in nested queries would give more expressive power to the language. Finally, the query language could be extended to become SQL compatible, thereby permitting to interact with the so-called “legacy systems”.

The result of a query, i.e., a set of objects, could be made persistent, thereby permitting users to group objects according to their interests.² Then, these sets could have associated indices to permit queries to be resolved against them, as

²This facility would correspond to the traditional notion of “views” found in object-oriented database systems.

provided by the ObjectStore system [34, 49] (Section 2.5). Moreover, other types of collections could be supported (in addition to sets), including lists and bags.

Currently, views correspond to schemas containing *entire* classes. A more fine-grained approach could be taken to permit the selection of *parts* of classes, i.e., a class selected by a view could have only some of its attributes, relationships and methods actually selected. This would give more flexibility to the system.

A critical issue in the design of database systems is the support for versioning control and schema evolution. These features are essential in modern information systems to cope with the fast evolution of real-life applications. Obviously, object engines can have a broader range of application if they support these features.

Easy interface for defining schemas is an important feature of database systems. For this purpose, a graphical user interface could be developed to capture schemas in a high-level fashion and then generate the corresponding meta-objects. Surely, this would simplify the generation of object engines and would increase productivity.

Distribution Support

Currently, contexts are global entities and isolated from each other. Contexts could be linked to each other to form networks of cooperating contexts, thereby providing for large-scale name space administration.

A very important issue in distributed systems that remain to be completely tackled by our architecture for object engines is authorisation/security/protection.

The introduction of views in the architecture aims at providing a starting point in this direction. This approach needs further development and effective implementation.

9.4.3 Applications

The object engine for bibliographical information constructed as demonstration application can be further developed: much more information can be maintained by the object engine, and many useful methods can be added to the classes. Thus, it can be a valuable tool for literature search, helping research activities in general, including business-oriented research. Some other areas of application which can be investigated include: travel agencies, management of computer resources, office documents (especially SGML and ODA documents) and department stores. Finally, because meta-data is part of the core of any CASE tool, we believe that Stabilis can be used to support management and generation of program code.

APPENDIX **A**

Object Model Definition

The object model concepts are formally defined according to the set and graph theories. The purpose of the formal definition is to provide a basis for implementing the object model concepts, as well as a basis for defining an *object manipulation language* that is independent of such an implementation.

Moreover, the formal definition permits the representation of the object model in terms of itself, i.e., the formal definition of the object model is *reflexive*. Thus, the formal definition also provides a basis for defining a model which permits us to represent information about schemas and objects. Such a model is referred to as *meta-object model* and it provides the basis for implementing the core of meta-information systems.

Tuple Notation

Given a tuple $t = (t_1, \dots, t_n)$ whose definition is a tuple (e_1, \dots, e_n) , let the notation $t.e_i$ denote t_i . For example, if $\alpha = (\text{age}, \text{Integer}, 0, \text{Person})$ is a tuple whose definition is (n, p, k, c) — defined below as *attribute specification* — then $\alpha.n$, $\alpha.p$, $\alpha.k$ and $\alpha.c$, respectively, denote **age**, **Integer**, **0** and **Person**.

Naming Assumptions

Let us assume the existence of the following countably infinite sets of names:

- set PN of primary type names
- set AN of attribute names
- set MN of method names
- set RN of role names
- set CN of class names
- set ON of object names

such that $PN \cap CN = \emptyset$.


A.1 Values, Types and Domains

Every attribute of every object has a value for which there is a textual representation, such as an integer or a string. For this reason, values have associated semantics which are well defined by primary types; each primary type stands for a set of values that have the same semantics. Such a set of values is the domain defined by a certain primary type. The domain of an attribute of a certain primary type is the domain defined by the primary type, i.e., the value assigned to an attribute of a certain primary type can vary only over the domain defined by the primary type.

As an example, let us suppose that a type named `Integer` defines the semantics for integer values, and a type named `String` defines the semantics for string values.


Then the domain of an attribute of type **Integer** includes $0, 1, 2, 3, \dots$, and the domain of an attribute of type **String** includes any delimited sequence of characters, such as **wet**, **wind**, **cool**, **freezing**, etc. An attribute of type **Integer** can be assigned the values $0, 1, 2, 3, \dots$, and an attribute of type **String** can be assigned the values **wet**, **wind**, **cool**, **freezing**, etc.

Therefore, let us assume the existence of values and value semantics, and define primary types and primary domains.

Notation A.1 The symbol \mathcal{V} denotes the set of all values. □ 

Definition A.1 (Primary Type) A primary type is a doublet (n, s) , where:

- $n \in PN$
- s denotes a value semantics □

Notation A.2 The symbol \mathcal{P} denotes the set of all primary types. □ 


Example A.1 We can have a type t_1 such that $t_1.n = \text{Integer}$ and $t_1.s$ denotes a semantics for integer values, and a type t_2 such that $t_2.n = \text{String}$ and $t_2.s$ denotes a semantics for string values. ◇

Type Identification

Primary types must be unambiguously identifiable by names in order to enable their use in textual notations, such as in specifications of attributes in class diagrams and in programs. Thus, a primary type name can be assigned to at most one primary type to ensure that there is an one-to-one mapping between primary type name and primary type.

Invariant A.1 (Primary Type Name Distinction) $\forall x, y \in \mathcal{P}$: if $x.n = y.n$ then $x = y$. ♦


Since each primary type has a distinct name, we define a notation to refer to primary types through their names.

 **Notation A.3** Given a primary type name $x \in PN$ and a primary type $p \in \mathcal{P}$ such that $p.n = x$, the notation π_x (type named x) denotes p . □


Example A.2 If we have a type t_1 such that $t_1.n = \text{Integer}$ then π_{Integer} denotes t_1 , and if we have a type t_2 such that $t_2.n = \text{String}$ then π_{String} denotes t_2 . ♦

Primary Domain

The domain defined by a primary type is the set of values that have the semantics defined by the type. We first define a notation to indicate that a value has the semantics defined by a primary type and then define primary domain.


 **Notation A.4** Given a value $v \in \mathcal{V}$ and a primary type $t \in \mathcal{P}$, the notation $v \dashv t$ (v is of type t) denotes that v has the semantics denoted by $t.s$. □


Example A.3 Let 5 be an integer value and let t_1 be a type such that $t_1.s$ denotes the semantics for integer values, then $5 \dashv t_1$ (5 is of type t_1). ♦

 **Definition A.2 (Primary Domain)** Given a primary type $t \in \mathcal{P}$, the primary domain with respect to t , denoted as $Dom(t)$, is the set of all values of type t :

$$Dom(t) = \{v \in \mathcal{V} \mid v \dashv t\}$$

Example A.4 If we have a type t_1 and the values 1, 3 and 5 such that $1 \vdash t_1$, $3 \vdash t_1$ and $5 \vdash t_1$, then $\text{Dom}(t) \supseteq \{1, 3, 5\}$. ◇

Notation A.5 The symbol \mathcal{D} denotes the set of all primary domains. □ 

Notation A.6 The symbol \mathbb{Z} denotes the primary domain containing all integer values. □ 

Typographic Convention

The context in which a value is inserted is normally sufficient to determine its type. Thus, as a typographic convention, we invariably represent values using **Sans serif** font and, conversely, everything represented in such font is a value. For example, a value that is the string containing only the character “7” and a value that is the integer number 7 are both represented as 7. In case of ambiguity, a string value is enclosed by quotes.

Names and Reflexivity

Primary type names, as any other name, are values, more specifically string values, as a requirement to ensure that the object model formal definition is reflexive¹. For example, the primary type names **Integer** and **String** are values and, accordingly, they are represented using **Sans serif** font.

¹The meta-object model assumes the existence of a primary type, named **String**, that defines the domain containing all string values.

A.2 Attribute

A.2.1 Attribute Specification

The specification of an attribute contains an attribute name, a primary type name, a Boolean value that indicates whether the attribute is or not a *key*, and a class name for the following reasons:

1. Each attribute of an object has a name which is distinct from the names of the other attributes of the object in order to permit each attribute to be referred to unambiguously with respect to the object.
2. Each attribute of an object has a value and a type specification for such value in order to provide a basis for a type checking mechanism, i.e., a mechanism that ensures that the value varies over only a certain primary domain.
3. Each attribute of an object is optionally defined as a key attribute, in which case it has to be indexed for query resolution purposes.
4. Attributes are specified by classes, i.e., every attribute specification is part of a class.

Definition A.3 (Attribute Specification) *An attribute specification is a tuple (n, p, k, c) , where $n \in AN$, $p \in PN$, $k \in \mathbb{Z}$ and $c \in CN$.* □

The integer value k in Definition A.3 corresponds to the Boolean value that indicates whether or not the attribute is key. For this reason, the value of k must be constrained to the values 0 and 1.

Invariant A.2 (Attribute Key Range) Let s be an attribute specification, then $0 \leq s.k \leq 1$. ♦

Example A.5 The following tuples are attribute specifications:

- (name, String, 1, Client)

The name of the attribute is **name**, its primary type is **String**, the attribute is key and it is part of the class **Client**.

- (balance, Integer, 0, Account)

The name of the attribute is **balance**, its primary type is **Integer**, the attribute is not key and it is part of the class **Account**. ◇

Set of Attribute Specifications

The attributes specified by a certain class must have distinct names in order to permit their unambiguous identification with respect to the class. Thus, a set of attribute specifications is consistent if all the attributes have distinct names.

Definition A.4 (Consistent Set of Attribute Specifications) A set of attribute specifications A is consistent iff $\forall x, y \in A : \text{if } x.n = y.n \text{ then } x = y$. □

Example A.6 The following is a consistent set of attribute specifications for a class named **Account**:

{ (number, Integer, 1, Account),
 (balance, Integer, 0, Account),
 (overdraft, Integer, 0, Account),
 (interest, String, 0, Account) }

Attribute Variable

Each instance of a class contains a set of attribute variables, such that there is an one-to-one correspondence between the elements of such a set and the elements of the set of attribute specifications of that class. That is, for each attribute specification in the class there is an attribute variable in the instance. Such an attribute variable consists of (1) a “copy” of the attribute specification and (2) a value of the primary type in the attribute specification. As discussed in Chapter 8, the information provided by an attribute specification is used for type checking and index update when objects are created, modified or deleted.

Definition A.5 (Attribute Variable) An attribute variable is a doublet (s, v) , where s is an attribute specification and v is a value of primary type such that $v \vdash \pi_{s,p}$.

□

Example A.7 The following tuples are attribute variables:

- $((\text{name, String, 1, Client}), \text{“Gustav Klimt”})$
 - An instance of the class named **Client** contains such attribute variable.
 - $(\text{name, String, 1, Client})$ is an attribute specification.
 - “Gustav Klimt” is a value of the type named **String**.
- $((\text{balance, Integer, 0, Account}), 8034)$
 - An instance of the class named **Account** contains such attribute variable.
 - $(\text{balance, Integer, 0, Account})$ is an attribute specification.

– 8034 is a value of the type named Integer.

◇

Set of Attribute Variables

The attributes of a certain object must have distinct names in order to permit their unambiguous identification with respect to the object. Thus, a set of attribute variables is consistent if all the attributes have distinct names.

Definition A.6 (Consistent Set of Attribute Variables) A set of attribute variables A is consistent iff $\forall x, y \in A : \text{if } x.s.n = y.s.n \text{ then } x = y$. □

Example A.8 The following is a consistent set of attribute variables for an object of class named Account:

$$\begin{aligned} &\{ ((\text{number}, \text{Integer}, 1, \text{Account}), 50298), \\ &\quad ((\text{balance}, \text{Integer}, 0, \text{Account}), 3980), \\ &\quad ((\text{overdraft}, \text{Integer}, 0, \text{Account}), 200), \\ &\quad ((\text{interest}, \text{String}, 0, \text{Account}), B) \} \end{aligned}$$

Integer Values and Reflexivity

The representation of Boolean values using integer values, such as the case of k in Definition A.3, aims at simplifying the reflexivity² of the formal definition of the object model. This simplification is appropriate because (1) Boolean values can

²The meta-object model assumes the existence of a primary type, named `Integer`, that defines the domain containing all integer values.

be represented through integer values without loss of genericity and (2) integer values are necessary to specify multiplicity in relationships.

A.3 Relationship Elements

Objects contain references to objects according to relationships specified by classes. In this Section we define the basic elements necessary to specify relationships in classes and to represent them in objects. Once classes and objects are formally defined, in Section A.8 we complement the definitions introduced in this Section.

A.3.1 Class Relationship

The information about a relationship between two classes is represented by a complementary pair of *relationship specifications*, one in each class. Thus, the set of relationship specifications of a class defines all relationships of the class.

Relationship Semantics

Every relationship between two classes has a semantics which is one of the following: association, loose aggregation or tight aggregation. An object, therefore, must represent relationship semantics accordingly in order to enable proper interpretation of its references. This is realised by associating a type to each relationship specification of a class. Such a type is then “copied” by every instance of the class.

In the case of an association, a relationship between two classes is symmetric, whereas in both cases of aggregation the relationship is asymmetric because one of the classes is the aggregate and the other class is the component. Thus, the set of all types of relationship specifications contains: (1) a type for aggregate classes in tight aggregations, (2) a type for component classes in tight aggregations, (3) a type for aggregate classes in loose aggregations, (4) a type for component class in loose aggregations and (5) a type for classes in associations.

Definition A.7 (Relationship Specification Type Set) *The relationship specification type set is the set of symbols $\mathfrak{R} = \{\text{PTAE}, \text{STAE}, \text{PLAE}, \text{SLAE}, \text{ASSE}\}$.* □



Relationship Specification

Each relationship specification must contain the necessary information to realise operations (creation, deletion and navigation) on object relationships in a consistent way. For this reason, a relationship specification contains: (1) the name of the local class (the class to each the specification belongs), (2) the role of the local class (local role), (3) the name of the related class, (4) the role of the related class (related role), (5) the multiplicity of the related class (minimum and maximum cardinality), (6) the type of the relationship specification (which defines the semantics of the relationship with respect to the local class), and (7) a Boolean value that indicates whether or not the relationship is key and must be indexed for query resolution purposes.

Definition A.8 (Relationship Specification) *A relationship specification is a tuple $(lc, lr, rc, rr, l, u, e, k)$, where $lc, rc \in CN$, $lr, rr \in RN$, $l, u \in \mathbb{Z}$, $e \in \mathfrak{R}$ and*

$k \in \mathbb{Z}$, such that $0 \leq l \leq u$.

Terminology: The pair (l, u) is a multiplicity, where l is the minimum cardinality and u is the maximum cardinality. \square

Example A.9 Let us consider two related classes named **Car** and **Wheel**, as illustrated in Figure A.1. The classes have a tight aggregation relationship where **Car** is the aggregate class and **Wheel** is the component class: (1) a single instance of **Wheel** is part of at most one instance of **Car**, (2) a single instance of **Car** contains a number of instances of **Wheel** varying from 0 to 4, (3) the role of **Car** in the relationship is **Vehicle**, (4) the relationship is key from **Wheel** to **Car**, (5) the relationship is not key from **Car** to **Wheel**.

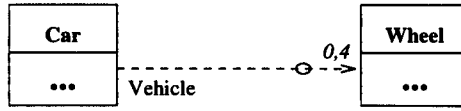


Figure A.1 Tight aggregation between classes **Car** and **Wheel**

Such relationship is represented by a pair of complementary relationship specifications such that one belongs to the class **Car** and the other one belongs to the class **Wheel**, respectively, defined by the following tuples:

- (Car, Vehicle, Wheel, Wheel, 0, 4, PTAE, 0)
- (Wheel, Wheel, Car, Vehicle, 0, 1, STAE, 1)

\diamond

The integer value k in Definition A.8 corresponds to the Boolean value that indicates whether or not the attribute is key. For this reason, the value of k must be constrained to the values 0 and 1.

Invariant A.3 (Relationship Key Range) Let s be a relationship specification, then $0 \leq s.k \leq 1$. ♦

Set of Relationship Specifications

Each element of the set of relationship specifications of a class needs to have an identification which is distinct from the identification of the others to permit each one to be referred to unambiguously with respect to the class. Although a convention for relationship identification which makes use of all information carried by relationship specifications can be elaborated, for simplicity, we will establish that all roles of the classes related to each class must be distinct. Also, this convention provides the basis for a simple notation in relationship operations, as discussed in Chapter 8. Thus, a set of relationship specifications is consistent if all its elements have distinct related roles.

Definition A.9 (Consistent Set of Relationship Specifications) A set of relationship specifications R is consistent iff $\forall x, y \in R : \text{if } x.rr = y.rr \text{ then } x = y$. □

Example A.10 Let us consider two related classes named **School** and **Person**, as illustrated in Figure A.2. Such classes have two associations: (1) an instance of **School** and a instance of **Person**, respectively, can be associated having the roles **Employer** and **Employee**, and (2) an instance of **School** and a instance of **Person**, respectively, can be associated having the roles **School** and **Student**.



Figure A.2 Associations between classes **School** and **Person**

*The class **School** has the following set of relationship specifications:*

{ (**School**, **Employer**, **Person**, **Employee**, 0, n, ASSE, 1),
 (**School**, **School**, **Person**, **Student**, 0, n, ASSE, 1) }

*Such set is consistent since all related roles **Employee** and **Student** are distinct.*

*The class **Person** has the following set of relationship specifications:*

{ (**Person**, **Employee**, **School**, **Employer**, 0, 1, ASSE, 1),
 (**Person**, **Student**, **School**, **School**, 0, n, ASSE, 1) }

*Such set is consistent since all related roles **Employer** and **School** are distinct.*

◇

A.3.2 Object Relationship

The relationships of a class specify the permitted relationships of its instances. (Figure A.3 illustrates the following discussion.) If a class *X* has a relationship *R* with a class *Y* then an object *x* that is an instance of *X* can have a relationship *r* corresponding to *R* with an object *y* that is an instance of *Y*.

A class relationship specifies a multiplicity for each class in the relationship. Such a multiplicity can specify any number of instances. For this reason, an

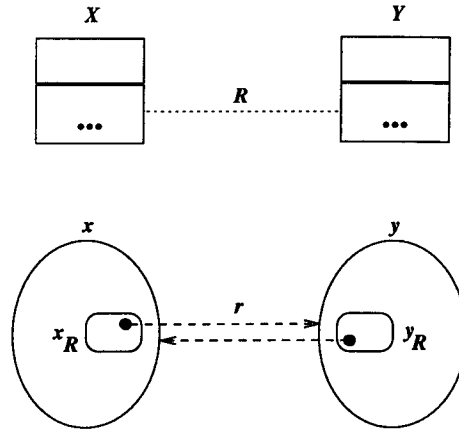


Figure A.3 Correspondence between related classes and related objects

instance of a class has a set of object references, or simply *reference set*, for every relationship of the class. Thus, x has a reference set x_R corresponding to R , and y has a reference set y_R corresponding to R .

A relationship between two objects is represented by a reference to each other in the corresponding reference sets. Thus, the relationship r between x and y is represented by a pair of complementary object references: (1) x has a reference to y in x_R and (2) y has a reference to x in y_R . For this reason, we say that a relationship is *bi-directional*.

Reference Set

Object references are realised through object names, which are unique to permit objects to be unambiguously referred to. Thus, reference sets are subsets of the set containing all object names.

Definition A.10 (Reference Set) Every finite subset of ON is a reference set. \square

Relationship Variable

Each instance of a class contains a set of relationship variables, such that there is an one-to-one correspondence between the elements of such a set and the elements of the set of relationship specifications of the class. That is, for each relationship specification in the class there is a relationship variable in the instance. Such a relationship variable consists of (1) a “copy” of the relationship specification and (2) a reference set. As discussed in Chapter 8, the information provided by a relationship specification is used for type checking, index update, navigation and automatic bi-directional relationship consistency.

Definition A.11 (Relationship Variable) *A relationship variable is a doublet (s, v) , where s is a relationship specification and v is a reference set.* \square

Example A.11 *Let us consider again the example of the classes named Car and Wheel, which have a tight aggregation, as illustrated in Figure A.1. Now, let us suppose that $\{i_0, i_1, i_2, \dots\}$ are object names, and that an instance of the class named Car, the object named i_0 , is aggregated with four instances of the class named Wheel, the objects named i_1, i_2, i_3 and i_4 , as illustrated in Figure A.4.*

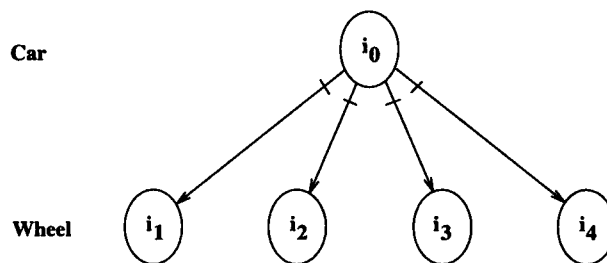


Figure A.4 Tight aggregation between instances of Car and Wheel

That aggregation is represented by the following relationship variables:

- ((Car, Vehicle, Wheel, Wheel, 0, 4, PTAE, 1), $\{i_1, i_2, i_3, i_4\}$)
 - The instance of the class Car contains such a relationship variable.
 - (Car, Vehicle, Wheel, Wheel, 0, 4, PTAE, 1) is a relationship specification.
 - $\{i_1, i_2, i_3, i_4\}$ is a reference set.
- ((Wheel, Wheel, Car, Vehicle, 0, 1, STAE, 0), $\{i_0\}$)
 - Each instance of the class Wheel contains such a relationship variable.
 - (Wheel, Wheel, Car, Vehicle, 0, 1, STAE, 0) is a relationship specification.
 - $\{i_0\}$ is a reference set. ◇

Relationship Variable Consistency

The multiplicity in the specification of a relationship variable defines the minimum and the maximum cardinality of the reference set in the variable.

Invariant A.4 (Relationship Variable Consistency) Let α be a relationship variable then $\alpha.s.l \leq |\alpha.v| \leq \alpha.s.u$. ◆

Example A.12 The reference set in a relationship variable specified as (Car, Vehicle, Wheel, Wheel, 0, 4, PTAE, 1) can contain a minimum of 0 and a maximum of 4 object references. ◇

Set of Relationship Variables

Each element of the set of relationship variables of a class needs to have an identification which is distinct from the identification of the others to permit each one to be referred to unambiguously with respect to the object. Since there is an one-to-one correspondence between the elements of a set of relationship variables of an object and the elements of a set of relationship specifications of a class, such a required distinction between relationship variables is simply obtained from the distinction between the corresponding relationship specifications. Thus, a set of relationship variables is consistent if all the relationships have distinct related roles.

Moreover, the semantics of tight aggregation enforces that an object is physically part of at most one object. In other words, if an object is the component in a certain tight aggregation then that object cannot be the component in another tight aggregation.

Definition A.12 (Consistent Set of Relationship Variables) *A set of relationship variables R is consistent iff:*

- (i) $\forall x, y \in R : \text{if } x.s.rr = y.s.rr \text{ then } x = y.$
- (ii) *If $\exists x, y \in R$ such that $x.s.e = \text{STAE}$ and $x.v \neq \emptyset$ and $y.s.e = \text{STAE}$ and $y.v \neq \emptyset$ then $x = y.$* □

Example A.13 *Let us consider the related classes named Door, Bolt and Window, as illustrated in Figure A.5. The class Bolt is the component class in two tight aggregations: (1) an instance of Bolt can be part of either an instance of Door or an instance of Window, (2) an instance of Door can aggregate from 0 to 4 instances*

of Bolt and (3) and instance of Window can aggregate from 2 to 8 instances of Bolt.

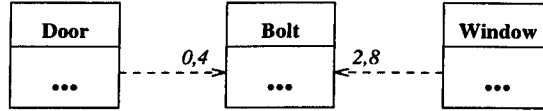


Figure A.5 Tight aggregations between classes Door, Bolt and Window

An instance of Bolt which is part of an instance of Door whose object name is i_7 has the following set of relationship variables:

$$\{ ((\text{Bolt}, \text{Bolt}, \text{Door}, \text{Door}, 0, 1, \text{STAE}, 1), \{i_7\}), \\ ((\text{Bolt}, \text{Bolt}, \text{Window}, \text{Window}, 0, 1, \text{STAE}, 1), \emptyset) \}$$

An instance of Bolt which is part of an instance of Window whose object name is i_8 has the following set of relationship variables:

$$\{ ((\text{Bolt}, \text{Bolt}, \text{Door}, \text{Door}, 0, 1, \text{STAE}, 1), \emptyset), \\ ((\text{Bolt}, \text{Bolt}, \text{Window}, \text{Window}, 0, 1, \text{STAE}, 1), \{i_8\}) \}$$

In both cases the set of relationship variables is consistent since the related roles are distinct (Door and Window) and there is only one non-empty reference set pertaining to a relationship variable whose specification contains the relationship type STAE. Now, let us suppose that an instance of Bolt has the following set of relationship variables:

$$\{ ((\text{Bolt}, \text{Bolt}, \text{Door}, \text{Door}, 0, 1, \text{STAE}, 1), \{i_7\}), \\ ((\text{Bolt}, \text{Bolt}, \text{Window}, \text{Window}, 0, 1, \text{STAE}, 1), \{i_8\}) \}$$

Such a set of relationship variables is not consistent since there are two non-empty reference sets pertaining to relationship variables whose specification contains the relationship type STAE. This inconsistency can be interpreted as the instance of Bolt being part of both an instance of Door and an instance of Window simultaneously, which does not comply with the semantics of tight aggregation.

◇

A.4 Class Elements

The definitions pertaining to *method* and *class*, respectively, presented in Section A.5 and Section A.6, depend on each other. Since such definitions come in that order, in this Section we anticipate part of the definitions pertaining to class, which do not depend on the definitions pertaining to method.

All definitions in this Section are illustrated using the class hierarchy with extents depicted in Figure A.6. The extent of each class is represented by an oval linked by a thick line to the corresponding class diagram. Each oval representing an extent contains smaller ovals which represent the objects in the extent.

Class

Each class has a name which is distinct from the name of any other class to permit each one to be referred to unambiguously. Moreover, every class can have a superclass. Thus, the definition of a class includes the name of the class and the name of its superclass, which can be a null name.

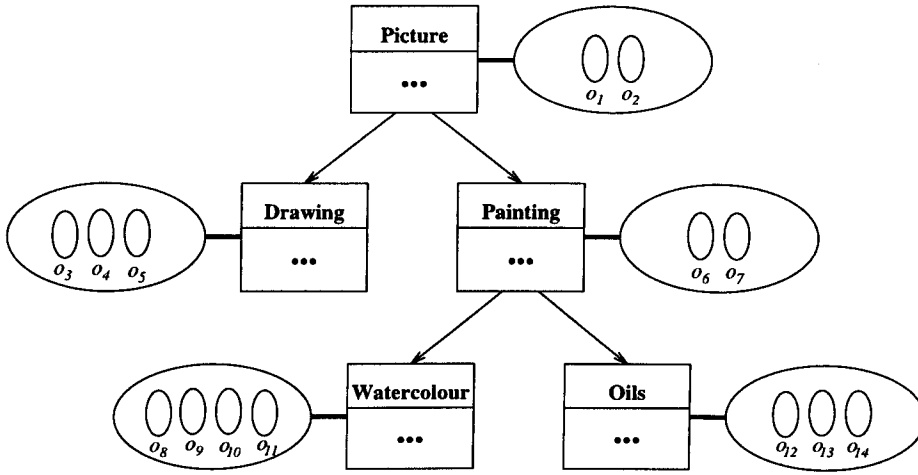


Figure A.6 Class hierarchy with extents

Notation A.7 The symbol \emptyset denotes a null name.



A class (Definition A.23) is a tuple, where two of the elements are:

- $n \in CN$
- $s \in (CN \cup \{\emptyset\})$

Example A.14 Each row of Table A.1 contains the elements n and s for a class shown in Figure A.6. ◇

We define a notation to denote the set of all classes in order to define formally that each class has a distinct name.

Notation A.8 The symbol \mathcal{C} denotes the set of all classes.



Invariant A.5 (Class Name Distinction) $\forall x, y \in \mathcal{C} : \text{if } x.n = y.n \text{ then } x = y.$ ◆

Since each class has a distinct name, we define a notation to refer to classes through their names.

n	s
Picture	\emptyset
Drawing	Picture
Painting	Picture
Watercolour	Painting
Oils	Painting

Table A.1 Example of class name and superclass name elements in classes

Notation A.9 Given a class name x and class c such that $c.n = x$, the notation κ_x denotes c . □

Example A.15 κ_{Picture} denotes the class named Picture. ◇

Object

Every object has a class path (a sequence of classes ordered according to their direct inheritance relation) and a most-specific class, which is the last element in the class path. Thus, the definition of an object includes a sequence of class names corresponding to the class path of the object and a class name corresponding to the name of the most-specific class of the object.

An *object* (Definition A.25) is a tuple, where two of the elements are:

- $c \in CN$
- φ is a sequence of class names in CN

Notation A.10 The symbol \mathcal{O} denotes the set of all objects. □

Example A.16 Each row of Table A.2 contains the elements c and \wp for an object shown in Figure A.6. ◇

object	c	\wp
o_1	Picture	$\langle \text{Picture} \rangle$
o_3	Drawing	$\langle \text{Picture}, \text{Drawing} \rangle$
o_6	Painting	$\langle \text{Picture}, \text{Painting} \rangle$
o_8	Watercolour	$\langle \text{Picture}, \text{Painting}, \text{Watercolour} \rangle$
o_{12}	Oils	$\langle \text{Picture}, \text{Painting}, \text{Oils} \rangle$

Table A.2 Example of class name and class path elements in objects

Class Extent and Direct Instance

The extent of a class α is the set of objects instantiated from α , that is, all objects whose most-specific class is α .

Definition A.13 (Class Extent) Given a class $c \in \mathcal{C}$, the extent of c , denoted as *Ext*(c), is the set of objects given by:

Ext

$$\text{Ext}(c) = \{o \in \mathcal{O} \mid c.n = o.c\}$$

An object that belongs to the extent of a class α is a direct instance of α .

Definition A.14 (Direct Instance) Given an object o and a class c , o is a direct instance of c if $o \in \text{Ext}(c)$. □

Class Deep Extent and Indirect Instance

The deep extent of a class α is the union of the extent of α and the extents of all subclasses of α . Therefore, the class path of any object in the deep extent of α contains α .

Definition A.15 (Class Deep Extent) Given a class $c \in \mathcal{C}$, the deep extent, denoted as $Ext^*(c)$, of c is the set of objects given by:

$$Ext^*(c) = \{o \in \mathcal{O} \mid c.n \in o.\varphi\}$$

An object that belongs to the extent of a subclass of a class α is an indirect instance of α . Thus, an indirect instance of a class α is any object in the deep extent of α that is not in the extent of α .

Definition A.16 (Indirect Instance) Given an object o and a class c , o is an instance of c if $o \in (Ext^*(c) \setminus Ext(c))$. □

Example A.17 Each row of Table A.3 contains the extent and the deep extent for a class shown in Figure A.6. ◇


Types and Domains

Since classes are abstract data types, we have that a type is either a primary type or a class. Moreover, the deep extent of a class α correspond to the domain of α . For the sake of simplicity, we define a notation to denote all types, and also we define a notation to refer to the domain of a type which applies to both primary types and classes.


class name	extent	deep extent
Picture	$\{o_1, o_2\}$	$\{o_1, \dots, o_{14}\}$
Drawing	$\{o_3, o_4, o_5\}$	$\{o_3, o_4, o_5\}$
Painting	$\{o_6, o_7\}$	$\{o_6, \dots, o_{14}\}$
Watercolour	$\{o_8, o_9, o_{10}, o_{11}\}$	$\{o_8, o_9, o_{10}, o_{11}\}$
Oils	$\{o_{12}, o_{13}, o_{14}\}$	$\{o_{12}, o_{13}, o_{14}\}$

Table A.3 Example of class extent and deep extent

Definition A.17 (Type) A type is either a primary type or a class. □

Notation A.11 The symbol TN denotes the set of all type names: 

$$TN = PN \cup CN$$

Definition A.18 (Type Domain) Given a type name $x \in TN$, the domain with respect to the type t such that $t.n = x$, denoted as $\xi(x)$, is given by: 

$$\xi(x) = \begin{cases} \text{Dom}(\pi_x) & \text{if } x \text{ is a primary type name,} \\ \text{Ext}^*(\kappa_x) & \text{if } x \text{ is a class name.} \end{cases}$$

Example A.18 Let us suppose that there is a primary type named `Integer`. Using the notation ξ to denote the domain of such primary type and the domain of the class named `Watercolour` in Figure A.6, respectively, we have the following equalities:

- $\xi(\text{Integer}) = \text{Dom}(\pi_{\text{Integer}})$
- $\xi(\text{Watercolour}) = \text{Ext}^*(\kappa_{\text{Watercolour}})$ ◇

A.5 Method

The state of an object is encapsulated by an interface composed of methods or operations which can be applied to the object. Such methods are specified and applied according to the following rules:

1. A method is specified in a class, i.e., a method is part of a class.
2. A method specified in a class α can be applied to any instance of α .
3. A method is part of one and only one class.
4. A method of a class α can be applied only to instances of α .
5. Every class specifies a set of methods.
6. Only methods of a class α can manipulate the state of an instance of α .

A method accepts a sequence of values and/or objects as arguments and returns another value or object as a result. The acceptable sequence of arguments is specified by a sequence of types, and the result is specified by another type. That is, each argument or result must belong to the domain of the corresponding type.

Moreover, there must be a means of identifying the methods of a certain class such that each method can be referred to unambiguously with respect to the class. For this purpose, we can simply establish a convention where by methods have names such that the name of each method of a class is distinct from the names of the other methods of the class. However, such a convention precludes that the methods of a class which have the same semantics and differ only with respect to their types of arguments and/or result have the same name. Therefore,

we establish a convention where by methods of a class can have the same name as long as their sequences of argument types and result type are different. Thus, every method is identified by a triple composed of a name, a sequence of argument types and a result type. Such a triple is referred to as the method signature.

Definition A.19 (Signature) A signature is a triple $s = (n, A, r)$, where:

- $n \in MN$
- A is a finite sequence of the form $\langle a_1, \dots, a_n \rangle$, where $\forall i \mid 1 \leq i \leq n : a_i \in TN$
- $r \in TN$ □

A method is implemented by a function that maps a product of source domains to a target domain, according to a semantics specified for the method: the first source domain is the extent of the class to which the method belongs, the remaining source domains are the argument domains, and the target domain is the result domain.

Definition A.20 (Method) A method is a tuple $m = (c, s, f : S \rightarrow T, b)$, where:

- $c \in CN$
- s is a signature
- $f : S \rightarrow T$ is a function mapping a product of source domains to a target domain, of the form:

$$f : Ext(\kappa_c) \times \xi(s.a_1) \times \dots \times \xi(s.a_n) \longrightarrow \xi(s.r)$$

- b denotes the semantics (behaviour) of function f □

Example A.19 Let us suppose that there are primary types named *Integer* and *String*, and that there are classes named *Person* and *School*. The following tuple is a method:

$(\text{School}, (\text{register_student}, \langle \text{Person}, \text{String} \rangle, \text{Integer}), f_1 : S \rightarrow T, b_1)$

- *School* is the class to which the method belongs.
- $(\text{register_student}, \langle \text{Person}, \text{String} \rangle, \text{Integer})$ is the method signature.
 - *register_student* is the method name.
 - $\langle \text{Person}, \text{String} \rangle$ is the sequence of argument types.
 - *Integer* is the result type.
- $f_1 : S \rightarrow T$ is a function that implements the method, and it has the following form:

$$f_1 : \text{Ext}(\kappa_{\text{School}}) \times \xi(\text{Person}) \times \xi(\text{String}) \longrightarrow \xi(\text{Integer})$$

- b_1 denotes the semantics of f_1 : For the instance of *School* to which the method is applied, register the instance of *Person* given as first argument as a student of the course whose name is given as second argument (*String*), and returns the registration number of the student as a result (*Integer*). \diamond

Set of Methods

Each element of the set of methods of a class needs to have an identification which is distinct from the identification of the others to permit each one to be referred to unambiguously with respect to the class. As previously discussed, such distinct identification is obtained through method signatures.

Definition A.21 (Consistent Set of Methods) A set of methods M is consistent iff

$\forall x, y \in M : \text{if } x.s = y.s \text{ then } x = y.$ \square

Example A.20 The following is a consistent set of methods for a class named School:

$$\begin{aligned} & \{ (\text{School}, (\text{register_student}, \langle \text{Person}, \text{String} \rangle, \text{Integer}), f_1 : S \rightarrow T, b_1), \\ & (\text{School}, (\text{register_student}, \langle \text{Person} \rangle, \text{Integer}), f_2 : S \rightarrow T, b_2), \\ & (\text{School}, (\text{issue_certificate}, \langle \text{Person} \rangle, \text{Integer}), f_3 : S \rightarrow T, b_3) \} \end{aligned}$$

Method Transformation and Inheritance Semantics

A class α defines a set of methods which can be applied to instances of α , and additionally α may inherit methods from a class β , the superclass of α . However, the methods of β are only applicable to instances of β , i.e., they cannot be applied to instances of α . For this reason, we define an operator to “transform” inherited methods such that they are applicable to instances of a subclass.

Moreover, an inherited method may either preserve the original method semantics or define a new one. If the method semantics is preserved then the semantics of the inheritance is *incorporation*, otherwise it is *substitution*.

Definition A.22 (Method Transformation) Given a method $m = (c, s, f, b)$ and a class name x , such that $\kappa_x.s = c$, the transformation of m with respect to the class named x , denoted as $\Gamma_x(m)$, is a tuple (x, s, f', b') , where:

- f' is a function of the form:

$$f' : \text{Ext}(\kappa_x) \times \xi(s.a_1) \times \cdots \times \xi(s.a_n) \longrightarrow \xi(s.r)$$

Γ

- b' is the semantics of f' , such that:

1. If $b' = b$ then $\Gamma_x(m)$ is an incorporation of m by κ_x

2. If $b' \neq b$ then $\Gamma_x(m)$ is a substitution of m by κ_x

□

Example A.21 Let us consider a class named **University** which inherits a method named `register_student` from a class named **School**, as illustrated in Figure A.7, where attributes and other methods are not shown.

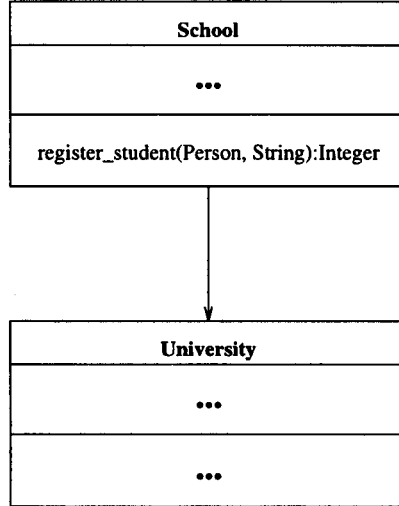


Figure A.7 Example of method inheritance

For simplicity of notation, let α and β , respectively, denote the classes named **University** and **School**, i.e., $\alpha = \kappa_{University}$ and $\beta = \kappa_{School}$. Also, let μ denote the method named `register_student`. The specification of μ can be given by:

$$\mu = (\text{School}, (\text{register_student}, \langle \text{Person}, \text{String} \rangle, \text{Integer}), f_1 : S \rightarrow T, b_1)$$

In such a specification of method μ , function $f_1 : S \rightarrow T$ has the following

form:

$$f_1 : Ext(\beta) \times \xi(Person) \times \xi(String) \longrightarrow \xi(Integer)$$

The transformation of method μ with respect to α , $\Gamma_{University}(\mu)$, can be given by one of the following equalities:

1. $\Gamma_{University}(\mu) = (University, (register_student, \langle Person, String \rangle, Integer), f_2 : S \rightarrow T, b_1)$

α incorporates μ : the original semantics of μ , i.e., b_1 , is preserved.

2. $\Gamma_{University}(\mu) = (University, (register_student, \langle Person, String \rangle, Integer), f_2 : S \rightarrow T, b_2)$

α substitutes μ : the original semantics of μ is substituted by the semantics b_2 .

In both cases, function $f_1 : S \rightarrow T$ specified for method μ is replaced by function $f_2 : S \rightarrow T$, which has the following form:

$$f_2 : Ext(\alpha) \times \xi(Person) \times \xi(String) \longrightarrow \xi(Integer)$$

Such function replacement enables method $\Gamma_{University}(\mu)$ be applied to instances of α , as the first source domain has changed from $Ext(\beta)$ to $Ext(\alpha)$. \diamond

We extend the use of the operator Γ for method transformation to sets of methods, i.e., if such operator is applied to a set of methods then all methods are transformed.

Notation A.12 Given a consistent set of methods $M = \{m_1, \dots, m_n\}$ and a class name x , the notation $\Gamma_x(M)$ denotes the transformation of M into $M' = \{m'_1, \dots, m'_n\}$, where $\forall i \mid 1 \leq i \leq n : m'_i = \Gamma_x(m_i)$. \square

A.6 Class

In this Section we complement the definitions pertaining to *class* presented in Section A.4, by making use of the definitions pertaining to *method* presented in Section A.5.

The complete definition of a class α consists of the following items:

1. The name of α .
 2. The name of the superclass of α or \emptyset if α has no superclass.
 3. A sequence of class names representing the class path of α . The sequence is ordered according to the inheritance relation between classes: the last element is the name of α and the first element is the name of the superclass of α that has no superclass. If α has a superclass, the sequence is obtained by adding the class name of α to the end of the sequence pertaining to the superclass, otherwise the sequence contains only the name of α . If α inherits from a class β , the name of α cannot belong to the sequence pertaining to β , otherwise there would be a “cycle” in the inheritance, i.e., α indirectly inherits from itself, which is not permitted.
 4. A consistent set of attributes locally defined by α .
 5. A consistent set of relationships locally defined by α .
-

6. A consistent set of methods locally defined by α .
7. A set containing the attributes locally defined by α and the attributes inherited by α , if any. Such a set must be consistent, i.e., all attributes in the set must have distinct names.
8. A set containing the relationships locally defined by α and the relationships inherited by α , if any. Such a set must be consistent, i.e., all relationships in the set must have distinct related role names.
9. A set containing the methods locally defined by α and the methods inherited by α , if any. Such a set must be consistent, i.e., all the methods in the set must have distinct signatures.

Definition A.23 (Class) A class is a tuple $c = (n, s, \varphi, PA, PR, PM, A, R, M)$, where:

- $n \in CN$
- $s \in (CN \cup \{\emptyset\})$
- φ is a sequence of class names in CN
- PA is a set of attribute specifications
- PR is a set of relationship specifications
- PM is a set of methods
- A is a set of attribute specifications
- R is a set of relationship specifications
- M is a set of methods

such that:

- (i) $\forall a \in PA : a.c = n$
- (ii) $\forall r \in PR : r.lc = n$
- (iii) $\forall m \in PM : m.c = n$
- (iv) PA is consistent
- (v) PR is consistent
- (vi) PM is consistent
- (vii) If $s = \emptyset$ then:

$$(a) \varphi = \{n\}$$

$$(b) A = PA$$

$$(c) R = PR$$

$$(d) M = PM$$

- (viii) If $s \neq \emptyset$ then:

$$(a) \varphi = \kappa_s.\varphi \cup \{n\}$$

$$(b) A = PA \cup \kappa_s.A$$

$$(c) R = PR \cup \kappa_s.R$$

$$(d) M = PM \cup \Gamma_n(\kappa_s.M)$$

$$(e) n \notin \kappa_s.\varphi$$

$$(f) \forall x \in PA : \forall y \in \kappa_s.A : x.n \neq y.n$$

$$(g) \forall x \in PR : \forall y \in \kappa_s.R : x.rr \neq y.rr$$

$$(h) \forall x \in PM : \forall y \in \kappa_s.M : x.s \neq y.s$$

Terminology:

- *PA is the partial set of attributes of c*
- *PR is the partial set of relationships of c*
- *PM is the partial set of methods of c*
- *A is the total set of attributes of c*
- *R is the total set of relationships of c*
- *M is the total set of methods of c*

□

The constraints on attribute names, relationship related roles and method signatures imposed in Definition A.23 naturally ensure the consistency of the total set of attributes, the total set of relationships and the total set of methods of a class.

Proposition A.1 (Class Consistency) $\forall c \in \mathcal{C}$:

- (i) *$c.A$ is consistent*
- (ii) *$c.R$ is consistent*
- (iii) *$c.M$ is consistent*

□

Example A.22 Let us consider the class hierarchy depicted in Figure A.8. For simplicity of notation, classes are named A, B, C, D, X and Y, attributes are denoted as a_1, \dots, a_5 , methods are denoted as m_1, \dots, m_5 , and relationships are denoted as r_1 and r_2 . If these labels are expanded then we can have, for example, that a_1 represents the attribute salary: Integer.

Each column of Table A.4 contains the specification for a class shown in Figure A.8. Also for simplicity of notation, the specifications of attributes, methods and relationships are denoted in the Table using their labels in the Figure. For example, if a_1 represents salary: Integer in the Figure then a_1 represents (salary, Integer, 0, A) in the Table. ◇

A.7 Object

An object o that is an instance of a class α consists of:

1. A set of attribute variables corresponding to the total set of attribute specifications of α .
2. A set of relationship variables corresponding to the total set of relationship specifications of α .
3. A set of methods corresponding to the total set of methods of α .

Every attribute/relationship of o is specified by an attribute/relationship of α , while the methods of o are “copies” of the methods of α . For this reason, we say that α *models* o , i.e., a class is a model of its instances. More specifically, every attribute specification of α models an attribute variable of o , and every relationship specification of α models a relationship variable of o . For methods, however, we simply say that every method of o is *equal* to a method of α . Therefore, we define an operator to represent the correspondence between an attribute/relationship specification with an attribute/relationship variable.

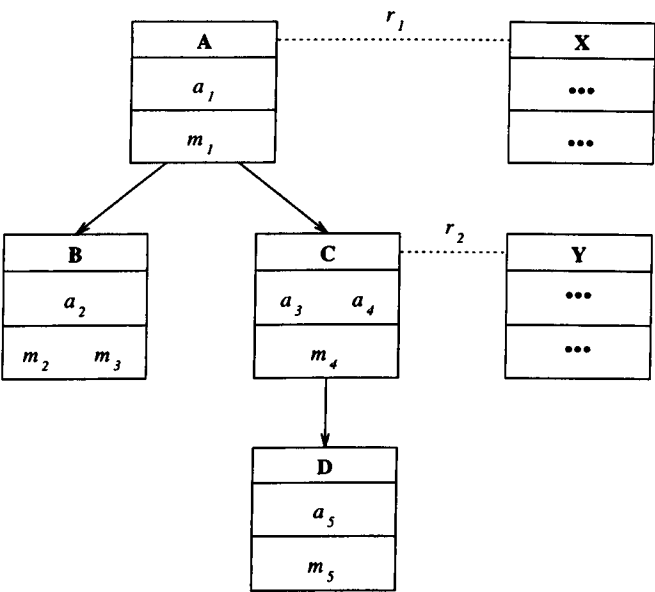


Figure A.8 Class hierarchy with labelled members

<i>n</i>	A	B	C	D
<i>s</i>	\emptyset	A	A	C
\wp	$\langle A \rangle$	$\langle A, B \rangle$	$\langle A, C \rangle$	$\langle A, C, D \rangle$
<i>PA</i>	$\{a_1\}$	$\{a_2\}$	$\{a_3, a_4\}$	$\{a_5\}$
<i>PR</i>	$\{r_1\}$	\emptyset	$\{r_2\}$	\emptyset
<i>PM</i>	$\{m_1\}$	$\{m_2, m_3\}$	$\{m_4\}$	$\{m_5\}$
<i>A</i>	$\{a_1\}$	$\{a_1, a_2\}$	$\{a_1, a_3, a_4\}$	$\{a_1, a_3, a_4, a_5\}$
<i>R</i>	$\{r_1\}$	$\{r_1\}$	$\{r_1, r_2\}$	$\{r_1, r_2\}$
<i>M</i>	$\{m_1\}$	$\{\Gamma_B(m_1), m_2, m_3\}$	$\{\Gamma_C(m_1), m_4\}$	$\{\Gamma_D(\Gamma_C(m_1)), \Gamma_D(m_4), m_5\}$

Table A.4 Example of class specification

Definition A.24 (Specification-Variable Correspondence) Given an attribute (a relationship) specification λ and an attribute (a relationship) variable x , λ models x , denoted as $\lambda \models x$, iff $x.s = \lambda$. \square

Example A.23 Let us consider the attribute specification λ_1 and the attribute variable x_1 given by:

$$\lambda_1 = (\text{name}, \text{String}, 1, \text{Client})$$

$$x_1 = ((\text{name}, \text{String}, 1, \text{Client}), \text{"Gustav Klimt"})$$

We have that $\lambda_1 \models x_1$ since $x_1.s = \lambda_1$.

Similarly, let us consider the relationship specification λ_2 and the relationship variable x_2 given by:

$$\lambda_2 = (\text{Car}, \text{Vehicle}, \text{Wheel}, \text{Wheel}, 0, 4, \text{PTAE}, 1)$$

$$x_2 = ((\text{Car}, \text{Vehicle}, \text{Wheel}, \text{Wheel}, 0, 4, \text{PTAE}, 1), \{i_1, i_2, i_3, i_4\})$$

We have that $\lambda_2 \models x_2$ since $x_2.s = \lambda_2$. \diamond

In addition to attribute variables, relationship variables and methods, an object o that is an instance of a class α also contains:

1. A unique name to permit o to be unambiguously referred to.
2. The name of α to permit type checking.
3. The set of class names that represents the class path of α to permit type checking.

Definition A.25 (Object) Given a class $t \in \mathcal{C}$, an object is a tuple (n, c, \wp, A, R, M) , where:

- $n \in ON$
- $c \in CN$
- \wp is a sequence of class names in CN
- A is a set of attribute variables
- R is a set of relationship variables
- M is a set of methods

such that:

- (i) $c = t.n$
- (ii) $\wp = t.\wp$
- (iii) $|A| = |t.A|$
- (iv) $\forall i \mid 1 \leq i \leq |A|, s_i \in t.A, v_i \in A : s_i \models v_i$
- (v) $|R| = |t.R|$
- (vi) $\forall i \mid 1 \leq i \leq |R|, s_i \in t.R, v_i \in R : s_i \models v_i$
- (vii) $M = t.M$

□

Example A.24 Let us consider the classes with corresponding instances depicted in Figure A.9. An object is represented by a rectangle with rounded corners and is connected to the respective class by a thick line. For simplicity, object names are denoted by integers. Also, the object diagrams do not show all details of attributes, relationships and methods. For example, the complete composition of an instance of the class named **Person** is illustrated in Table A.5. ◇

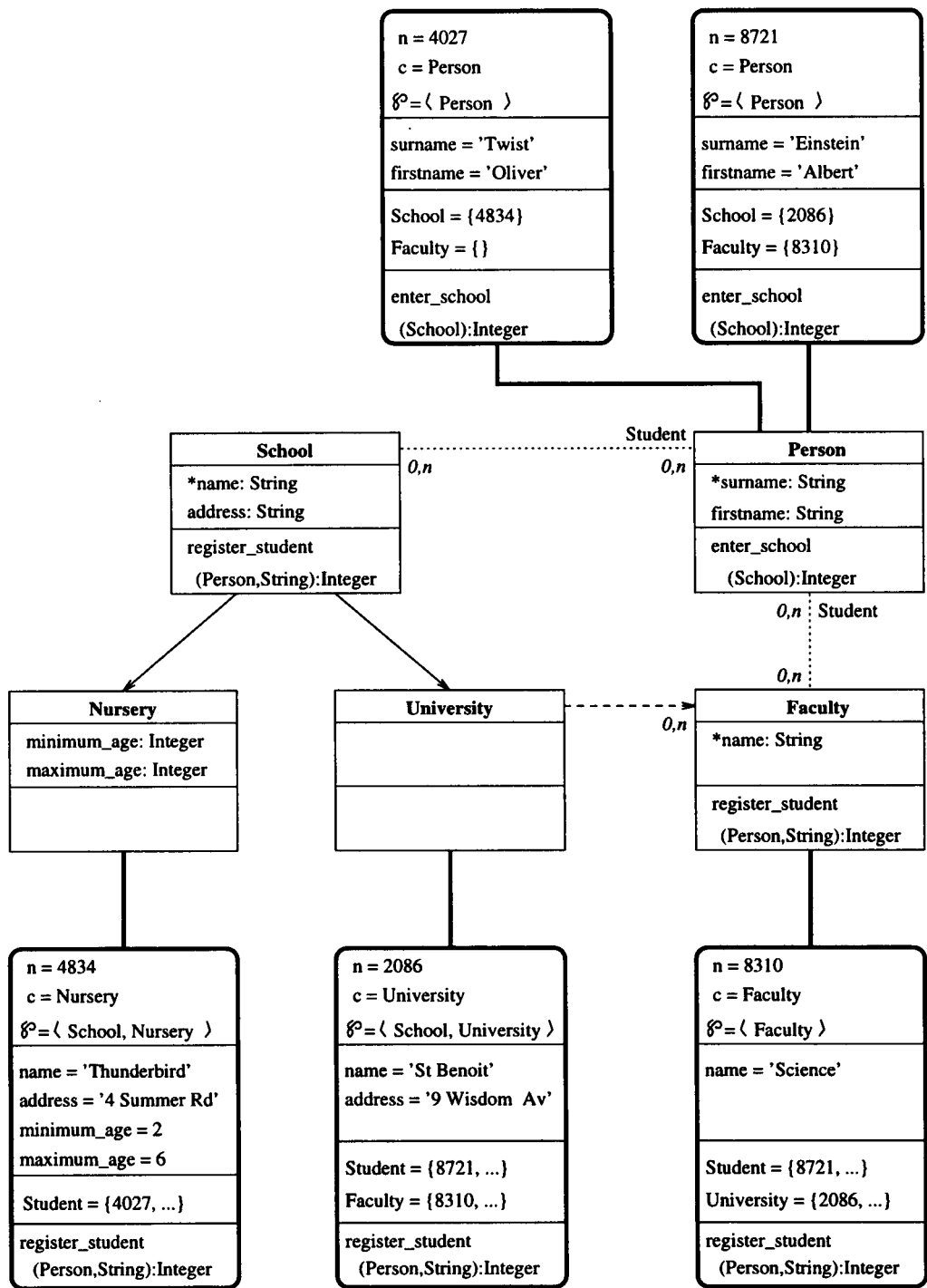


Figure A.9 Correspondence between class and instance elements

n	8721
c	Person
\wp	$\langle \text{Person} \rangle$
A	$\{ ((\text{surname}, \text{String}, 1, \text{Person}), \text{Einstein}) ,$ $((\text{firstname}, \text{String}, 0, \text{Person}), \text{Albert}) \}$
R	$\{ ((\text{Person}, \text{Student}, \text{School}, \text{School}, 0, n, \text{ASSE}, 1), \{ 2086 \}),$ $((\text{Person}, \text{Student}, \text{Faculty}, \text{Faculty}, 0, n, \text{ASSE}, 1), \{ 8310 \}) \}$
M	$\{ (\text{Person}, (\text{enter_school}, \langle \text{School} \rangle, \text{Integer}), f : S \rightarrow T, b) \}$

Table A.5 Example of object elements

Object Invariants and Notation

The set of all objects is equal to the union of the extents of all classes since every object is an instance of a class.

Proposition A.2 (Instantiation) $\mathcal{O} = \bigcup_{c \in \mathcal{C}} \text{Ext}(c)$ □

The name of each object o must be unique to permit o to be unambiguously referred to.

Invariant A.6 (Object Name Uniqueness) $\forall x, y \in \mathcal{O} : \text{if } x.n = y.n \text{ then } x = y.$ ◆

We define a notation to refer to objects through their names since each object has a unique name.

Notation A.13 Given an object name i , the notation Θ_i denotes the object o such that $o.n = i$. □



Example A.25 Θ_{8721} denotes the object shown in Table A.5. ◇

Also, we define a notation to denote the value of an attribute variable of an object through the name of the variable.

Notation A.14 Given an object o and an attribute variable $\alpha \in o.A$, the notation $o \dashrightarrow \alpha.s.n$ denotes $\alpha.v$. □

Example A.26 Let us suppose that o denotes the object shown in Table A.5. Then, $o \dashrightarrow \text{surname}$ denotes Einstein, and $o \dashrightarrow \text{firstname}$ denotes Albert. ◇

Object State and Interface

The state of an object o is composed by the attributes and the relationships of o , while the interface of o is defined by the methods of o .

Definition A.26 (Object State) Given an object o , the state of o is the doublet $(o.A, o.R)$. □

Definition A.27 (Object Interface) Given an object o , the interface of o is $o.M$. □

The consistency of the sets of attributes, relationships and methods of an object is naturally ensured since classes model objects.

Proposition A.3 (Instance Consistency) $\forall o \in \mathcal{O}$:

- (i) $o.A$ is consistent
- (ii) $o.R$ is consistent
- (iii) $o.M$ is consistent □

A.8 Relationship

In this Section we complement the definitions pertaining to *relationships* presented in Section A.3, by making use of the definitions pertaining to *classes* and *objects*, presented in Section A.6 and Section A.7, respectively.

Related Classes

A class relationship always involves a pair of classes. Consequently, if a class α contains a relationship specification σ then there must exist a class β that contains a relationship specification λ which is complementary to σ . We must note, however, that there is no constraint to force α and β to be distinct, i.e., α and β can be the same class. Moreover, the types of σ and λ must reflect the relationship semantics.

Invariant A.7 (Related Class) $\forall \alpha \in \mathcal{C} : \forall \sigma \in \alpha.PR : \exists \beta \in \mathcal{C}$ such that:

$$(i) \ \sigma.rc = \beta.n$$

$$(ii) \ \exists \lambda \in \beta.PR \text{ such that:}$$

$$(a) \ \lambda.rc = \alpha.n$$

$$(b) \ \sigma.lr = \lambda.rr$$

$$(c) \ \lambda.lr = \sigma.rr$$

$$(d) \ \text{If } \sigma.e = \text{PTAE then } \lambda.e = \text{STAE}$$

$$(e) \ \text{If } \sigma.e = \text{PLAE then } \lambda.e = \text{SLAE}$$

$$(f) \ \text{If } \sigma.e = \text{ASSE then } \lambda.e = \text{ASSE}$$

Terminology:

- the pair (σ, λ) defines a class relationship between α and β
 - σ and λ are complementary relationship specifications
 - if $\sigma.e = \text{PTAE}$ or $\sigma.e = \text{PLAE}$ then:
 - α is the left class in the relationship
 - β is the right class in the relationship
 - the class relationship is an aggregation
 - α is the aggregate class
 - β is the component class
 - $\sigma.lr$ is the aggregate role
 - $\lambda.lr$ is the component role
 - $\sigma.l$ is the component minimum cardinality
 - $\sigma.u$ is the component maximum cardinality
 - $\lambda.l$ is the aggregate minimum cardinality
 - $\lambda.u$ is the aggregate maximum cardinality
 - if $\sigma.e = \text{PTAE}$ then:
 - the class relationship is a tight aggregation
 - α is the parent tight aggregation class
 - σ is the parent tight aggregation specification
 - β is the sibling tight aggregation class
-

- λ is the sibling tight aggregation specification
- if $\sigma.e = \text{PLAE}$ then:
 - the class relationship is a loose aggregation
 - α is the parent loose aggregation class
 - σ is the parent loose aggregation specification
 - β is the sibling loose aggregation class
 - λ is the sibling loose aggregation specification
- if $\sigma.e = \text{ASSE}$ then:
 - the class relationship is an association
 - α and β are associated classes
 - σ and λ are association entries
 - if α is the left class in the relationship then:
 - * β is the right class in the relationship
 - * $\sigma.lr$ is the left role
 - * $\lambda.lr$ is the right role
 - * $\sigma.l$ is the right minimum cardinality
 - * $\sigma.u$ is the right maximum cardinality
 - * $\lambda.l$ is the left minimum cardinality
 - * $\lambda.u$ is the left maximum cardinality



Example A.27 Let us consider a simple hypertext model where links between nodes have semantics to permit their organisation as documents, according to the following rules:

- L_1 . A document is a set of linked nodes.
- L_2 . A node may be part of several documents concurrently.
- L_3 . A document has one node designated as the root node.
- L_4 . A node may be root of at most one document.
- L_5 . Nodes may be arranged as a hierarchical structure.
- L_6 . Node hierarchies may interleave with each other.
- L_7 . A node may contain a reference to any other node, including itself.
- L_8 . A node may be a note about other node.

Such hypertext model is implemented by the classes depicted in Figure A.10, where relationships are labelled r_1, \dots, r_5 , and have the following semantics:

- r_1 permits to define the set of nodes of a document (L_1) and permits to define a node as part of several documents (L_2) since it is a loose aggregation.
 - r_2 permits to define the root node of a document (L_3) and ensures that a node may be the root of at most one document (L_4) since it is a tight aggregation.
 - r_3 permits to represent hierarchical arrangement of nodes (L_5), possibly with interleaving between hierarchies (L_6) since it is a loose aggregation.
 - r_4 permits to represent references between nodes (L_7).
-

- r_5 permits to define a node as a note about other node (L_8).

The pairs of complementary relationship specifications corresponding to r_1, \dots, r_5 are shown in Table A.6, where each pair is denoted by σ and λ . For simplicity of notation, the class named **Document** is denoted by α , and the class named **Node** is denoted by β . \diamond

Related Objects

Object relationships are always bi-directional. (Figure A.11 illustrates the following discussion.) Consequently, if an object x contains a name i in the set of references of a relationship variable σ then there must exist an object y whose name is i and that contains the name of x (say j) in the set of references of a relationship variable λ whose specification is complementary to the specification of σ ; we say that σ and λ are complementary relationship variables.

Invariant A.8 (Related Objects) $\forall x \in \mathcal{O} : \forall \sigma \in x.R : \forall i \in \sigma.v : \exists y \in \mathcal{O}$ such that:

1. $y = \Theta_i$
2. $\exists \lambda \in y.R$ such that:

(a) $\lambda.s$ is complementary to $\sigma.s$

(b) $x.n \in \lambda.v$

Terminology:

- the pair (σ, λ) defines an object relationship between x and y

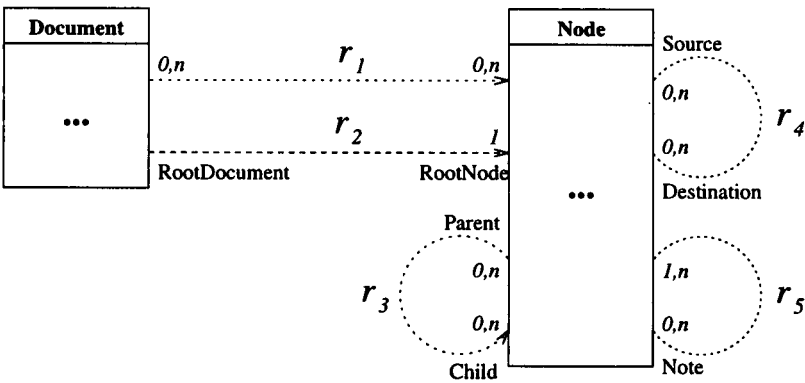


Figure A.10 Relationships between classes for a simple hypertext model

label	class	relationship specification
r_1	α	$\sigma = (\text{Document}, \text{Document}, \text{Node}, \text{Node}, 0, n, \text{PLAE}, 1)$
	β	$\lambda = (\text{Node}, \text{Node}, \text{Document}, \text{Document}, 0, n, \text{SLAE}, 1)$
r_2	α	$\sigma = (\text{Document}, \text{RootDocument}, \text{Node}, \text{RootNode}, 1, 1, \text{PTAE}, 1)$
	β	$\lambda = (\text{Node}, \text{RootNode}, \text{Document}, \text{RootDocument}, 0, 1, \text{STAE}, 1)$
r_3	β	$\sigma = (\text{Node}, \text{Parent}, \text{Node}, \text{Child}, 0, n, \text{PLAE}, 1)$
	β	$\lambda = (\text{Node}, \text{Child}, \text{Node}, \text{Parent}, 0, n, \text{SLAE}, 1)$
r_4	β	$\sigma = (\text{Node}, \text{Source}, \text{Node}, \text{Destination}, 0, n, \text{ASSE}, 1)$
	β	$\lambda = (\text{Node}, \text{Destination}, \text{Node}, \text{Source}, 0, n, \text{ASSE}, 1)$
r_5	β	$\sigma = (\text{Node}, \text{Node}, \text{Node}, \text{Note}, 0, n, \text{ASSE}, 1)$
	β	$\lambda = (\text{Node}, \text{Note}, \text{Node}, \text{Node}, 1, n, \text{ASSE}, 1)$

Table A.6 Example of complementary relationship specifications

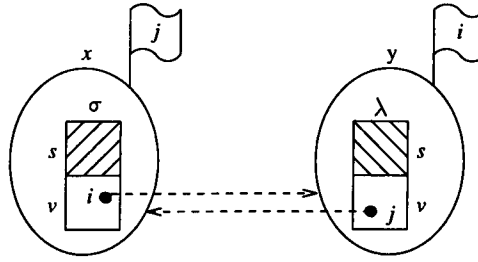


Figure A.11 Complementary relationship variables in related objects

- σ and λ are complementary relationship variables
- x and y are related with respect to the class relationship $(\sigma.s, \lambda.s)$
- $\sigma.s.lr$ is the role of x in the relationship
- $\lambda.s.lr$ is the role of y in the relationship
- if $\sigma.e = \text{PTAE}$ then the object relationship is a tight aggregation
- if $\sigma.e = \text{PLAE}$ then the object relationship is a loose aggregation
- if $\sigma.s.e = \text{PTAE}$ or $\sigma.s.e = \text{PLAE}$ then:
 - x and y are aggregated to each other
 - x is the aggregate object
 - y is the component object
- if $\sigma.e = \text{ASSE}$ then:
 - the object relationship is an association
 - x and y are associated objects



We define a notation to denote object relationship.

Notation A.15 Given two objects x and y , and two role names A and B , the notation $x \xrightleftharpoons[A]{B} y$ denotes a relationship between x and y , where A is the role of x and B is the role of y . \square

Example A.28 Let us consider an object x and an object y which are instances of the class named *Node* shown in Figure A.11, such that x is a parent node of y , that is, x and y are related with respect to the class relationship r_3 .

For simplicity of notation, let j denote the name of x ($x.n = j$) and i denote the name of y ($y.n = i$). Thus, the set of relationship variables of x contains an element σ and the set of relationship variables of y contains an element λ given by:

$$\sigma = ((\text{Node}, \text{Parent}, \text{Node}, \text{Child}, 0, n, \text{PLAE}, 1), \{i, \dots\})$$

$$\lambda = ((\text{Node}, \text{Child}, \text{Node}, \text{Parent}, 0, n, \text{SLAE}, 1), \{j, \dots\})$$

The role of x is *Parent* while the role of y is *Child*. Thus, using the notation for related objects, we have that:

$$x \xrightleftharpoons[\text{Parent}]{\text{Child}} y$$

Aggregate and Component Objects Distinction

The semantics of aggregation establishes that a component object is *part of* an aggregate object. Therefore, both objects are necessarily distinct.

Invariant A.9 (Aggregate and Component Objects Distinction) $\forall x, y \in \mathcal{O}$: if x and y are aggregated to each other then $x \neq y$. \blacklozenge

Relationship Variable Identification

We introduce a theorem which to justify aspects of the object manipulation language and algorithms, used in Chapter 8, pertaining to operations for creation and removal of object relationships .

Both relationship creation and removal operations are applied to an object x with only two parameters:

P_1 : An object y .

P_2 : The role of y in the relationship.

Such an operation affects a pair of complementary relationship variables: a variable σ in the corresponding set of x and a variable λ in the corresponding set of y . We must recall that the set of relationship variables of an object is consistent, i.e., all related roles are distinct from each other to permit variables to be identified (Definition A.9). Thus, the variable σ is directly identified by P_2 . The variable λ , however, is indirectly identified by information provided by σ : the remote role in λ is equal to the local role in σ . If the classes of x and y are related then λ exists. However, it must be shown that a relationship variable of y whose related role is equal to the local role of σ is necessarily λ , i.e., the complementary relationship variable of σ .

Example A.29 Let us consider an object x and an object y which are instances of the class named **Node** shown in Figure A.11. If an operation is applied to x to create a relationship with y (P_1), such that the role of y is **Child** (P_2), then the identification of the affected complementary relationship variables is proceeded according to the following steps:

1. A relationship variable σ is identified in the corresponding set of x by the related role **Child**. Such variable is given by:

$$\sigma = ((\text{Node}, \text{Parent}, \text{Node}, \text{Child}, 0, n, \text{PLAE}, 1), \{\dots\})$$

2. From σ , we have that the local role of x is **Parent**. Thus, a relationship variable λ is identified in the corresponding set of y by the related role **Parent**. One such variable is given by:

$$\lambda = ((\text{Node}, \text{Child}, \text{Node}, \text{Parent}, 0, n, \text{SLAE}, 1), \{\dots\})$$

Obviously, if all the elements of σ and λ are compared, we deduce that they are complementary. However, we want to prove that such a comparison is not necessary, i.e., only the fact that the remote role of λ (**Parent**) is equal to the local role of σ is sufficient. \diamond

Since complementary relationship variables have complementary relationship specifications and the set of relationship variables of an object o is modelled by the set of relationship specifications of the class of o , we have that σ and λ are complementary if their corresponding relationship specifications in the classes of x and y are complementary.

Theorem A.1 (Complementary Relationship Specifications) *Given the relationship specifications σ and λ , and the classes $\alpha, \beta \in \mathcal{C}$, such that $\sigma \in \alpha.PR$ and $\lambda \in \beta.PR$, σ and λ are complementary if:*

$$(i) \ \sigma.rc = \beta.n$$

$$(ii) \ \sigma.lr = \lambda.r$$

■

Proof: Let σ and λ be relationship specifications, and $\alpha, \beta \in \mathcal{C}$ be classes such that:

$$\sigma \in \alpha.PR$$

$$\lambda \in \beta.PR$$

$$\sigma.rc = \beta.n$$

$$\sigma.lr = \lambda.rr$$

From A.1 and Invariant A.7 (Related Classes) we have that $\exists \zeta \in \beta.PR$ such that:

$$\zeta.rc = \alpha.n$$

$$\zeta.lr = \sigma.rr$$

$$\zeta.rr = \sigma.lr$$

From A.1 and A.1 we have that:

$$\zeta.rr = \lambda.rr$$

From Proposition A.1 (Class Consistency) we have that $\beta.PR$ is consistent. Thus, from Definition A.9 (Consistent Set of Relationship Specifications), we have that:

$$\forall x, y \in \beta.PR : x.rr = y.rr \Rightarrow x = y$$

Since $\zeta \in \beta.PR$, from A.1, A.1 and A.1 we have that:

$$\zeta = \lambda$$

From A.1, A.1 and A.1 we have that:

$$\lambda.rc = \alpha.n$$

$$\lambda.lr = \sigma.rr$$

Therefore, from A.1, A.1, A.1, A.1, A.1, A.1 and Invariant A.7 (Related Classes), we have that σ and λ are complementary. \boxtimes

A.9 Single Inheritance

According to Definition A.23 (Class), a class may have a superclass; if a class d is derived from a base class b then $d.s = b.n$. Such a relation between b and d implies inheritance of attributes, methods and relationships of b by d .

Definition A.28 (Inheritance Arc) Given a pair of classes $b, d \in CN$, there exists an inheritance arc from b to d , denoted as Υ_d^b , iff $d.s = b.n$. \square

Example A.30 Figure A.12 illustrates an inheritance arc from a class b , named Picture, to a class d , named Drawing. \diamond

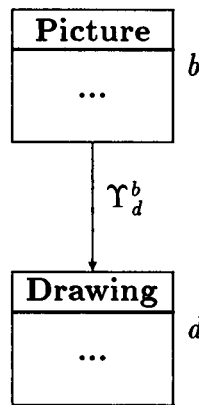


Figure A.12 Example of inheritance arc

We define a symbol to denote the set of all inheritance arcs.

Notation A.16 The symbol \mathcal{A} denotes the set of all inheritance arcs between classes in \mathcal{C} :

 \mathcal{A}

$$\mathcal{A} = \{\Upsilon_d^b \mid b, d \in \mathcal{C}\}$$

Since Definition A.23 establishes that a class has *at most one* superclass, only single inheritance is permitted.

Proposition A.4 (Single Inheritance) Let x, y, d be classes in \mathcal{C} . If $\exists \Upsilon_d^x, \Upsilon_d^y \in \mathcal{A}$ then $\Upsilon_d^x = \Upsilon_d^y$ and $x = y$. □

Tree-based Arrangement of Classes

Classes and corresponding inheritance arcs can be represented by a directed graph; every vertex of the graph represents a class and every arc (directed edge) of the graph represents an inheritance arc. We will prove that such a graph is a directed tree.³ Firstly, we define a notation for graphs which is summarised in Figure A.13 through examples.

Notation A.17 A directed graph G is denoted by a doublet (V, A) , where V and A , respectively, denote the set of vertices and the set of arcs of G . □

Notation A.18 Given a directed graph G and a vertex $v \in G.V$, the notation $\text{Indegree}(v)$ denotes the number of arcs in $G.A$ which have v as their final vertex. □

Indegree

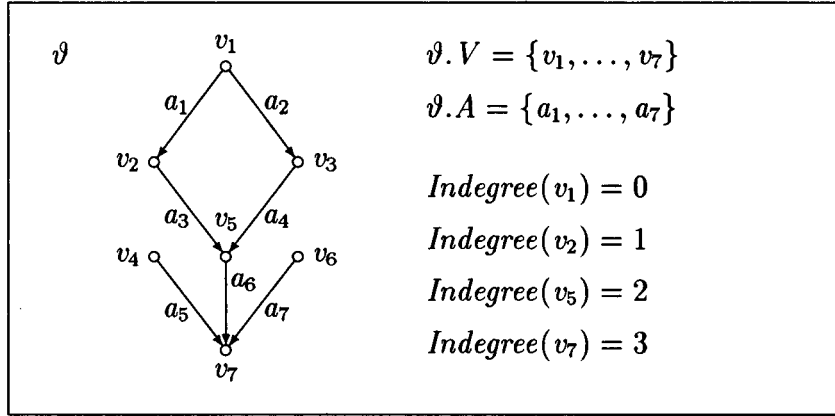
Notation A.19 Given a directed tree Ψ , the notation $\text{Root}(\Psi)$ denotes the vertex

Root

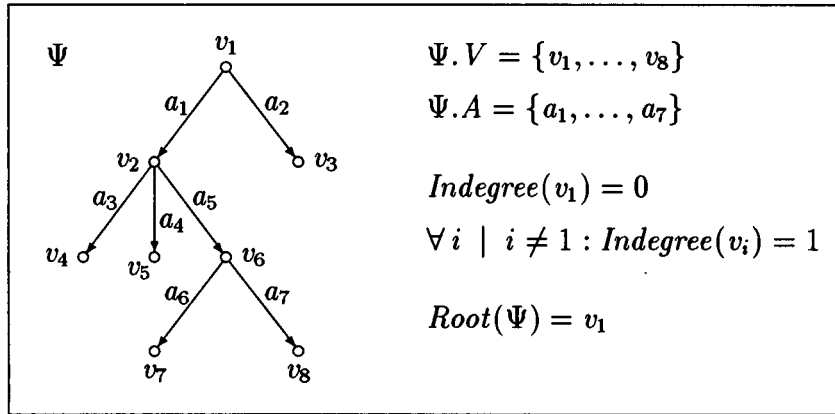
³A directed tree is also called an *arborescence* in the literature.

in $\Psi.V$ which is the root of Ψ .

□



(a) A directed graph ϑ



(b) A directed tree Ψ

Figure A.13 Example of graph notation

Also, we define a notation to denote the graph that represents all classes and inheritance arcs and, finally, prove that such classes are arranged as directed trees.

Notation A.20 The symbol \mathcal{G} denotes a directed graph such that $\mathcal{G}.V = \mathcal{C}$ and

$$\mathcal{G}.A = \mathcal{A}. \quad \square$$

Theorem A.2 (Tree-based Arrangement of Classes) *Every connected subgraph of \mathcal{G} is a directed tree.* ■

Proof: From Definition A.23 we have that a class has at most one superclass, thereby:

$$\forall c \in \mathcal{G}.V : \text{Indegree}(c) \leq 1$$

Let Ψ be a connected subgraph of \mathcal{G} , and let ν and α , respectively, be the number of vertices and the number of arcs in Ψ . Since Ψ is connected we have that $\alpha \geq \nu - 1$ (Theorem 5C in [67]). Let us assume that $\alpha = \nu - 1$, which then implies Ψ is a tree (Theorem 9A in [67]). Let $r = \text{Root}(\Psi)$, then:

$$\forall c \in \Psi.V \mid c \neq r : \text{Indegree}(c) = 1$$

Also, from Definition A.23 we have that:

$$\forall c \in \Psi.V : r.n \in c.\wp$$

Now, let us suppose that $x, y \in \Psi.V$ and that an arc Υ_y^x should be added to $\Psi.A$.

We have two cases to consider:

1. If $\boxed{y = r}$ then, from Definition A.23, we have that the addition of Υ_y^x to $\Psi.A$ is only possible if $r.n \notin x.\wp$, which contradicts A.1.
2. If $\boxed{y \neq r}$ then, from A.1, we have that the addition of Υ_y^x to $\Psi.A$ implies $\text{Indegree}(y) = 2$, which contradicts A.1.

Therefore, the arc Υ_y^x cannot be added to $\Psi.A$, which implies Ψ is a directed tree. ⊠

A.10 Class Hierarchy

According to Theorem A.2, \mathcal{G} is a directed acyclic graph (DAG). However, graph \mathcal{G} is not necessarily connected. More specifically, every (*connected*) *component* of \mathcal{G} is a distinct directed tree, which we call *class hierarchy*.

Definition A.29 (Class Hierarchy) A class hierarchy is a directed tree of classes that is a maximum subgraph of \mathcal{G} with respect to the connectedness property. \square

Since \mathcal{C} and \mathcal{A} are finite sets, from A.2 we have that the number of class hierarchies in \mathcal{G} is finite.

Corollary A.1 (Arrangement of Classes in Hierarchies) Graph \mathcal{G} is a finite set of class hierarchies. \square

Example A.31 The graph \mathcal{G} depicted in Figure A.14 is composed of class hierarchies Ψ_1, \dots, Ψ_5 . Vertices are labelled as c_i to denote classes and, for simplicity, arcs are not labelled. \diamond

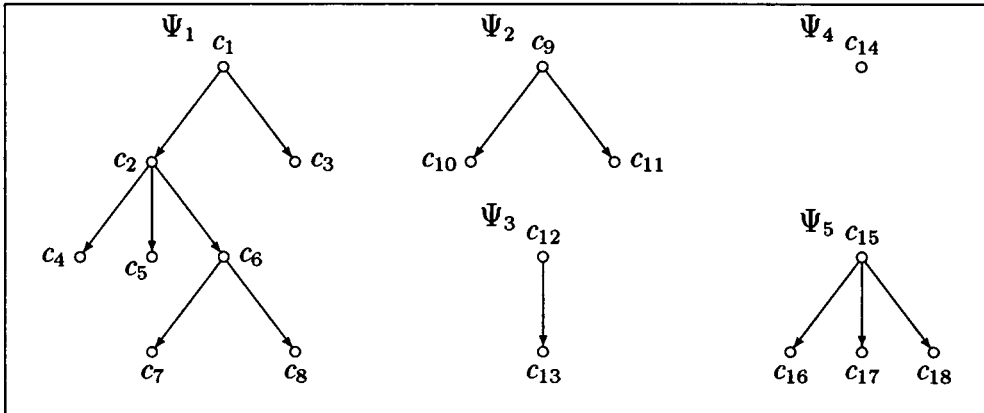


Figure A.14 Example of graph \mathcal{G}

Class Path

Since a class hierarchy is a directed tree, there is a unique *elementary path*⁴ from the root to every vertex of a class hierarchy. The sequence of classes (vertices) in such a path is called *class path*.

Definition A.30 (Class Path) Given a class hierarchy Ψ in \mathcal{G} and a class $c \in \Psi.V$, the class path with respect to c , denoted as $Path(c)$, is the sequence of classes in the elementary path from $Root(\Psi)$ to c , inclusive. □

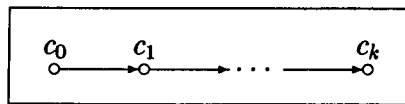
Example A.32 Let us consider the class hierarchy Ψ_1 depicted in Figure A.14. Some of the class paths in Ψ_1 is given by:

$$Path(c_1) = \langle c_1 \rangle$$

$$Path(c_4) = \langle c_1, c_2, c_4 \rangle$$

$$Path(c_7) = \langle c_1, c_2, c_6, c_7 \rangle$$

The class path with respect to a class c is represented by the sequence of class names given by $c.\varphi$. Let us consider the classes c_0, c_1, \dots, c_k such that c_{i+1} is direct subclass of $c_i \forall i, 0 \leq i \leq k-1$, which is represented as follows.



Then, $\forall j, 0 \leq j \leq k$, we have that:

⁴A elementary path is a sequence of arcs where the final vertex of one is the initial vertex of the next one such that the same vertex is not used more than once in the path.

1. $Path(c_j) = \langle c_0, \dots, c_j \rangle$
2. $c_j.\varphi = \langle c_0.n, \dots, c_j.n \rangle$

A.11 Class Conformity

According to Definition A.23 (*Class*), a class d that is derived from a base class b (i.e., $d.s = b.n$) contains all specifications (total sets of attributes, relationships and methods) of b and, possibly, adds new specifications. For this reason, we say that “ d conforms to b ”.

Intuitively, the specifications of a class x are contained in all *direct* and *indirect* subclasses of x . Consequently, any direct or indirect subclass of x conforms to x . In general, given a class x and a class y , if either $y = x$ or y is subclass of x then y conforms to x . In both cases, for simplicity, we say that “ y is a x ”. This *is-a* relation between x and y can be expressed in terms of the class path: y is a x if x is in the class path with respect to y .

Definition A.31 (*Is-a Relation*) Given the classes $x, y \in \mathcal{C}$, y is a x , denoted as $y \leq_\tau x$, iff $x.n \in y.\varphi$. □

Example A.33 Let us consider the classes *School*, *Nursery* and *University* in Figure A.9. The class *Nursery* contains the attributes *name* and *address*, the relationship with class *Person* having related role *Student* and the method *register_student*, which are specified by class *School*. For this reason, class *Nursery* conforms to class *School*. Similarly, class *University* conforms to class *School*. For simplicity

of notation, let x , y and z , respectively, denote the classes School, Nursery and University. Thus, we have that:

1. $(x.n = \text{School}) \wedge (x.\wp = \langle \text{School} \rangle) \Rightarrow x.n \in x.\wp \Rightarrow x \leq_\tau x$
2. $(x.n = \text{School}) \wedge (y.\wp = \langle \text{School}, \text{Nursery} \rangle) \Rightarrow x.n \in y.\wp \Rightarrow y \leq_\tau x$
3. $(x.n = \text{School}) \wedge (z.\wp = \langle \text{School}, \text{University} \rangle) \Rightarrow x.n \in z.\wp \Rightarrow z \leq_\tau x \diamond$

Deep Extent Reduction

Given a class x and a class y such that y is subclass of x , while the specification of y includes the specification of x , the deep extent of y is included in the deep extent of x . In other words, a subclass enlarges the specification and reduces the deep extent of its superclass.

Proposition A.5 (Deep Extent Reduction) *Given a class x and a class y , $y \leq_\tau x$ iff $\text{Ext}^*(y) \subseteq \text{Ext}^*(x)$.* □

Let us consider the classes c_0, c_1, \dots, c_k such that $\text{Path}(c_k) = \langle c_0, c_1, \dots, c_k \rangle$. Figure A.15 illustrates how class specification is enlarged and class deep extent is reduced from c_0 to c_k . We have that:

1. $c_k \leq_\tau c_{k-1} \leq_\tau \dots \leq_\tau c_0$
2. $\text{Ext}^*(c_k) \subseteq \text{Ext}^*(c_{k-1}) \subseteq \dots \subseteq \text{Ext}^*(c_0)$

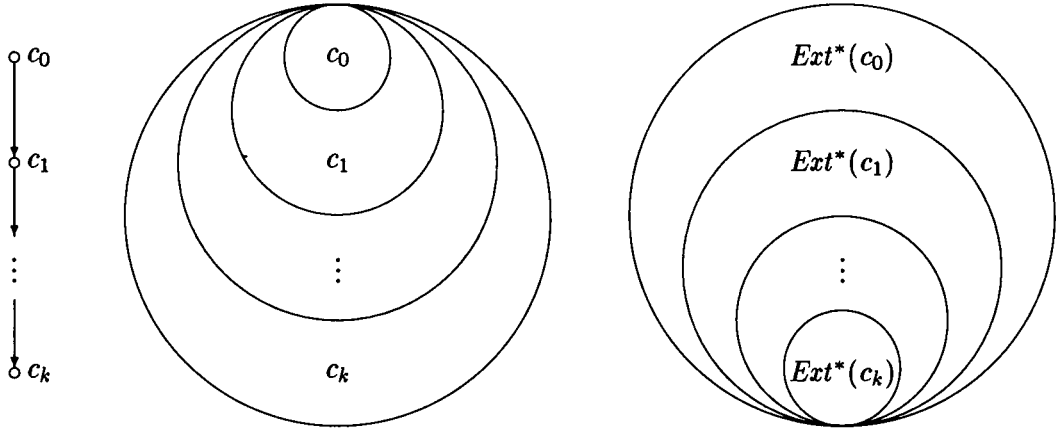


Figure A.15 Specification enlargement and deep extent reduction in class path

A.12 Partial Ordering of Classes

The *is-a* relation defines a *partial order* in \mathcal{C} , that is, \mathcal{C} is a poset (partially ordered set) with respect to the *is-a* relation.

Theorem A.3 (Partial Order Relation between Classes) *The is-a relation has the following properties:*

- (i) *Reflexive property:* $\forall a \in \mathcal{C} : a \leq_{\tau} a$
- (ii) *Antisymmetric property:* $\forall a, b \in \mathcal{C} : \text{if } a \leq_{\tau} b \text{ and } b \leq_{\tau} a \text{ then } a = b$
- (iii) *Transitive property:* $\forall a, b, c \in \mathcal{C} : \text{if } a \leq_{\tau} b \text{ and } b \leq_{\tau} c \text{ then } a \leq_{\tau} c$ ■

Proof: Let a , b and c be classes in \mathcal{C} .

- (i) *Reflexive property:*

From Definition A.23 (Class) we have that $a.n \in a.\wp$.

From Definition A.31 (Is-a Relation) we have that $a.n \in a.\wp \Rightarrow a \leq_{\tau} a$.

(ii) *Antisymmetric property:*

Let us suppose that $a \leq_\tau b$ and $b \leq_\tau a$. From Definition A.31 and Definition A.23 we have that:

$$\begin{aligned} a \leq_\tau b \Rightarrow b.n \in a.\wp &\Rightarrow \begin{cases} \text{either } a = b & (1A) \\ \text{or } b \text{ is superclass of } a & (1B) \end{cases} \\ b \leq_\tau a \Rightarrow a.n \in b.\wp &\Rightarrow \begin{cases} \text{either } a = b & (2A) \\ \text{or } a \text{ is superclass of } b & (2B) \end{cases} \end{aligned}$$

Thus, one combination in the product $\{1A, 1B\} \times \{2A, 2B\}$ holds. If the combination (1A, 2A) holds then $a = b$. According to Theorem A.2 (Tree-based Arrangement of Classes) any other combination is absurd. Therefore:

$$a \leq_\tau b \wedge b \leq_\tau a \Rightarrow a = b$$

(iii) *Transitive property:*

Let us suppose that $a \leq_\tau b$ and $b \leq_\tau c$. According to Definition A.31 and Theorem A.2 we have that:

$$\begin{aligned} a \leq_\tau b \Rightarrow b.n \in a.\wp &\Rightarrow b.\wp \preceq a.\wp \\ b \leq_\tau c \Rightarrow c.n \in b.\wp &\Rightarrow c.\wp \preceq b.\wp \end{aligned}$$

From A.1 and A.1 we have that:

$$c.\wp \preceq a.\wp$$

From Definition A.23 we have that:

$$c.n \in c.\wp$$

From A.1, A.1 and Definition A.31 we have that:

$$c.n \in a.\wp \Rightarrow a \leq_{\tau} c$$

Therefore:

$$a \leq_{\tau} b \wedge b \leq_{\tau} c \Rightarrow a \leq_{\tau} c$$

Corollary A.2 (Partial Ordering of Classes) *The set of classes \mathcal{C} is partially ordered with respect to the is-a relation.* □

APPENDIX B

Theorem Proof

In this Appendix we prove the Theorem 5.1 (*Root Subtree Self-containment*), which is stated in Chapter 5.

Theorem: *A subtree H of a class hierarchy Ψ in \mathcal{G} is self-contained iff H is a root-subtree.* ■

Firstly, we recall that if a class x is (directly or indirectly) subclass of a class y then $x \leq_\tau y$ and, conversely, if $x \leq_\tau y$ then x is (directly or indirectly) subclass of y . For this reason, we can also define class hierarchy self-containment using the *is-a* relation: for every class x in a set of classes C that is self-contained with respect to hierarchy we have that if there is class y such that $x \leq_\tau y$ then y belongs to C , and vice-versa.

Proposition B.1 (Partial Order and Hierarchy Self-Containment) *A set of classes C is self-contained with respect to hierarchy iff $\forall x \in C : \forall y \in \mathcal{C} : \text{if } x \leq_\tau y \text{ then } y \in C$.* □

Proof:

(i) H is self-contained $\Rightarrow H$ is a root-subtree

Since H is self-contained we have that:

$$\forall y \in H.V : \forall x \in \mathcal{C} : y \leq_{\tau} x \Rightarrow x \in H.V$$

Since $\Psi.V \subseteq \mathcal{C}$, from B.1 we have that:

$$\forall y \in H.V : \forall x \in \Psi.V : y \leq_{\tau} x \Rightarrow x \in H.V$$

Since Ψ is a class hierarchy we have that:

$$\forall y \in \Psi.V : y \leq_{\tau} \text{Root}(\Psi)$$

Since H is subtree of Ψ , from B.1 we have that:

$$\forall y \in H.V : y \leq_{\tau} \text{Root}(\Psi)$$

From B.1 and B.1 we have that:

$$\text{Root}(\Psi) \in H.V$$

Therefore:

$$\text{Root}(H) = \text{Root}(\Psi)$$

(ii) H is a root-subtree $\Rightarrow H$ is self-contained

Since H is a root-subtree we have that:

$$\text{Root}(H) = \text{Root}(\Psi)$$

Let $y \in H.V$ and $x \in \mathcal{C}$ be classes such that:

$$y \leq_{\tau} x$$

From B.1 we have that:

$$x.\wp \preceq y.\wp$$

Since $y \in H.V$ we have that:

$$\text{Root}(H) \in y.\wp$$

From B.1 and B.1 we have that:

$$\text{Root}(\Psi) \in y.\wp$$

From B.1 and B.1 we have that:

$$\text{Root}(\Psi) \in x.\wp$$

From B.1 we have that:

$$x \in \Psi.V$$

From B.1 and B.1 we have that:

$$x \in H.V$$

Therefore:

$$\forall y \in H.V : \forall x \in \mathcal{C} : y \leq_\tau x \Rightarrow x \in H.V$$

APPENDIX C

Meta-classes Definition

In this Appendix we give a formal definition for all meta-classes.

Naming Assumptions

All class and schema names used in the meta-schema are considered as “reserved”, i.e., user-defined schemas cannot contain such names. Moreover, the only primary types used in the schema correspond to the domains integer and strings. Thus, let us assume the existence of the following sets:

- a set \mathcal{R}_{PN} of reserved primary type names
- a set \mathcal{R}_{PT} of reserved primary types
- a set \mathcal{R}_{PD} of reserved primary domains
- a set \mathcal{R}_{AN} of attribute names
- a set \mathcal{R}_{CN} of reserved class names
- a set \mathcal{R}_{WN} of reserved schema names

such that:

-
- (i) $PN \supseteq \mathcal{R}_{PN}$
 - (ii) $\mathcal{P} \supseteq \mathcal{R}_{PT}$
 - (iii) $\mathcal{D} \supseteq \mathcal{R}_{PD}$
 - (iv) $AN \supseteq \mathcal{R}_{AN}$
 - (v) $CN \supseteq \mathcal{R}_{CN}$
 - (vi) $WN \supseteq \mathcal{R}_{WN}$
 - (vii) $\mathcal{R}_{PN} = \{\text{Integer}, \text{String}\}$
 - (viii) $\mathcal{R}_{PT} = \{\pi_{\text{Integer}}, \pi_{\text{String}}\}$
 - (ix) $\mathcal{R}_{PD} = \{\text{Dom}(\pi_{\text{Integer}}), \text{Dom}(\pi_{\text{String}})\}$
 - (x) $\pi_{\text{Integer}}.s$ is a semantics for integer values
 - (xi) $\pi_{\text{String}}.s$ is a semantics for string values
 - (xii) $\text{Dom}(\pi_{\text{Integer}})$ is the set of integer values
 - (xiii) $\text{Dom}(\pi_{\text{String}})$ is the set of string values
 - (xiv) $\mathcal{R}_{AN} = \{$
 name, key, signature,
 aggregate_role, aggregate_min_card, aggregate_max_card,
 component_role, component_min_card, component_max_card,
 left_role, left_min_card, left_max_card,
 right_role, right_min_card, right_max_card
 }
 - (xv) $\mathcal{R}_{CN} = \{$
 Class, Attribute, Method, Relationship, Schema,
-

IntegerAttribute, StringAttribute,
 Aggregation, LooseAggregation, TightAggregation, Association
 }

(xvi) $\mathcal{R}_{WN} = \{\text{Meta, Class, Attribute, Method, Relationship, Schema}\}$

Meta-classes

According to the meta-schema depicted in Figure 6.1, the meta-classes are formally defined as follows.

Invariant C.1 (Meta-class Class) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{Class}$, then:

- (1) $c.s = \emptyset$
- (2) $c.PA = \{(\text{name, String, 1, Class})\}$
- (3) $c.PR = \{$
 (Class, Class, Attribute, Attribute, 0, n, PTAE, 1),
 (Class, Class, Method, Method, 0, n, PTAE, 1),
 (Class, LeftClass, Relationship, LeftRelationship, 0, n, PLAE, 1),
 (Class, RightClass, Relationship, RightRelationship, 0, n, PLAE, 1),
 (Class, SuperClass, Class, SubClass, 0, n, ASSE, 1),
 (Class, SubClass, Class, SuperClass, 0, 1, ASSE, 1),
 (Class, RootClass, Schema, RootSchema, 0, n, SLAE, 1),
 (Class, NonRootClass, Schema, NonRootSchema, 0, n, SLAE, 1)
 }



Invariant C.2 (Meta-class Attribute) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{Attribute}$, then:

- (1) $c.s = \emptyset$
- (2) $c.PA = \{$
 $(\text{name}, \text{String}, 1, \text{Attribute}),$
 $(\text{key}, \text{Integer}, 0, \text{Attribute})$
 $\}$
- (3) $c.PR = \{(\text{Attribute}, \text{Attribute}, \text{Class}, \text{Class}, 0, 1, \text{STAE}, 1)\}$ ♦

Invariant C.3 (Meta-class StringAttribute) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{StringAttribute}$, then:

- (1) $c.s = \text{Attribute}$
- (2) $c.PA = \emptyset$
- (3) $c.PR = \emptyset$ ♦

Invariant C.4 (Meta-class IntegerAttribute) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{IntegerAttribute}$, then:

- (1) $c.s = \text{Attribute}$
- (2) $c.PA = \emptyset$
- (3) $c.PR = \emptyset$ ♦

Invariant C.5 (Meta-class Method) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{Method}$, then:

- (1) $c.s = \emptyset$
-

$$(2) \ c.PA = \{(\text{signature}, \text{String}, 1, \text{Method})\}$$

$$(3) \ c.PR = \{(\text{Method}, \text{Method}, \text{Class}, \text{Class}, 0, 1, \text{STAE}, 1)\}$$

◆

Invariant C.6 (Meta-class Relationship) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{Relationship}$, then:

$$(1) \ c.s = \emptyset$$

$$(2) \ c.PA = \{ \\ (\text{left_key}, \text{Integer}, 0, \text{Relationship}), \\ (\text{right_key}, \text{Integer}, 0, \text{Relationship}) \\ \}$$

$$(3) \ c.PR = \{ \\ (\text{Relationship}, \text{LeftRelationship}, \text{Class}, \text{LeftClass}, 1, 1, \text{SLAE}, 1), \\ (\text{Relationship}, \text{RightRelationship}, \text{Class}, \text{RightClass}, 1, 1, \text{SLAE}, 1) \\ \}$$

◆

Invariant C.7 (Meta-class Aggregation) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{Aggregation}$, then:

$$(1) \ c.s = \text{Relationship}$$

$$(2) \ c.PA = \{ \\ (\text{aggregate_role}, \text{String}, 1, \text{Aggregation}), \\ (\text{component_role}, \text{String}, 1, \text{Aggregation}), \\ (\text{component_min_card}, \text{Integer}, 0, \text{Aggregation}), \\ (\text{component_max_card}, \text{Integer}, 0, \text{Aggregation}) \\ \}$$

$$(3) \ c.PR = \emptyset$$

◆

Invariant C.8 (Meta-class LooseAggregation) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{LooseAggregation}$, then:

$$(1) \ c.s = \text{Aggregation}$$

$$(2) \ c.PA = \{ \\ \quad (\text{aggregate_min_card}, \text{Integer}, 0, \text{Aggregation}), \\ \quad (\text{aggregate_max_card}, \text{Integer}, 0, \text{Aggregation}) \\ \}$$

$$(3) \ c.PR = \emptyset$$

◆

Invariant C.9 (Meta-class TightAggregation) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{TightAggregation}$, then:

$$(1) \ c.s = \text{Aggregation}$$

$$(2) \ c.PA = \emptyset$$

$$(3) \ c.PR = \emptyset$$

◆

Invariant C.10 (Meta-class Association) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{Association}$, then:

$$(1) \ c.s = \text{Relationship}$$

$$(2) \ c.PA = \{ \\ \quad (\text{left_role}, \text{String}, 1, \text{Association}), \\ \quad (\text{right_role}, \text{String}, 1, \text{Association}), \\ \quad (\text{left_min_card}, \text{Integer}, 0, \text{Association}), \\ \}$$

$(\text{right_min_card}, \text{Integer}, 0, \text{Association}),$
 $(\text{left_max_card}, \text{Integer}, 0, \text{Association}),$
 $(\text{right_max_card}, \text{Integer}, 0, \text{Association}),$
 $\}$

(3) $c.PR = \emptyset$



Invariant C.11 (Meta-class Schema) Let $c \in \mathcal{C}$ be a meta-class such that $c.n = \text{Schema}$, then:

(1) $c.s = \emptyset$

(2) $c.PA = \{(\text{name}, \text{String}, 1, \text{Schema})\}$

(3) $c.PR = \{$

$(\text{Schema}, \text{RootSchema}, \text{Class}, \text{RootClass}, 0, 1, \text{PLAE}, 1),$

$(\text{Schema}, \text{NonRootSchema}, \text{Class}, \text{NonRootClass}, 0, n, \text{PLAE}, 1),$

$(\text{Schema}, \text{SuperSchema}, \text{Schema}, \text{SubSchema}, 0, n, \text{PLAE}, 1),$

$(\text{Schema}, \text{SubSchema}, \text{Schema}, \text{SuperSchema}, 0, n, \text{SLAE}, 1)$


$\}$



APPENDIX D


Meta-object Mapping

In this Appendix we show how classes, attributes, methods, relationships and schemas are mapped to meta-objects by using the formalisation presented in Appendix A and Chapter 5. All definitions and examples presented in this Appendix are equivalent to the definitions and examples presented in Chapter 6. Thus, they are illustrated using the schema shown in Figure 6.2.

Definition D.1 (Class Meta-object) Given a class name $n \in CN$ such that $\exists \kappa_n \in \mathcal{C}$, the class meta-object with respect to κ_n , denoted as $\varphi(n)$, is $m \in \xi(\text{Class})$ such that $m \dashrightarrow \text{name} = n$. φ 

Example D.1 Let us consider the class named **Person**. The notation $\varphi(\text{Person})$ denotes the instance y of the meta-class **Class** such that $y \dashrightarrow \text{name} = \text{Person}$. \diamond

Attribute Mapping

Notation D.1 Given a primary type name $p \in PN$, the notation $\text{AttCN}(p)$ denotes a class name in \mathcal{R}_{CN} as follows. AttCN 

- $\text{AttCN}(\text{Integer}) = \text{IntegerAttribute}$

- $AttCN(String) = StringAttribute$ □

Meta_A *Definition D.2 (Attribute Mapping)* The set of meta-objects that maps an attribute specification $\alpha = (n, p, k, c)$, denoted as $Meta_A(\alpha)$, is the set containing only the meta-objects $x \in \xi(Attribute)$, $y \in \xi(Class)$ such that:

$$(i) \ x.c = AttCN(\alpha.p)$$

$$(ii) \ x \dashrightarrow name = \alpha.n$$

$$(iii) \ x \dashrightarrow key = \alpha.k$$

$$(iv) \ y = \varphi(\alpha.c)$$

$$(v) \ x \underset{Attribute}{\overset{Class}{\rightleftharpoons}} y$$

□

Example D.2 Let us consider the attribute **surname** of class **Person**. A specification α for such attribute is given by:

$$\alpha = (surname, String, 1, Person)$$

Thus, $Meta_A(\alpha) = \{x, y\}$ such that:

$$x.c = StringAttribute$$

$$x \dashrightarrow name = surname$$

$$x \dashrightarrow key = 1$$

$$y = \varphi(Person)$$

$$x \underset{Attribute}{\overset{Class}{\rightleftharpoons}} y$$

Method Mapping

StrSig *Notation D.2* Given a method μ , the notation $StrSig(\mu)$ denotes the string obtained by concatenating the strings $\mu.s.n, \mu.s.a_1, \dots, \mu.s.a_n, \mu.s.r$ in that order and separating them using commas. □

Definition D.3 (Method Mapping) The set of meta-objects that maps a method μ , denoted as $Meta_M(\mu)$, is the set containing only the meta-objects $x \in \xi(\text{Method})$, $y \in \xi(\text{Class})$ such that:

Meta_M

- (i) $x \dashrightarrow \text{signature} = \text{StrSig}(\mu)$
- (ii) $y = \varphi(\mu.c)$
- (iii) $x \xrightleftharpoons[\text{Method}]{\text{Class}} y$

□

Example D.3 Let us consider the method `register_student` of class `School`. A specification μ for such method is given by:

$$\mu = (\text{School}, (\text{register_student}, \langle \text{Person}, \text{String} \rangle, \text{Integer}), f : S \rightarrow T, b)$$

Thus, $Meta_M(\mu) = \{x, y\}$ such that:

$$x.c = \text{Method}$$

$$x \dashrightarrow \text{signature} = \text{"register_student, Person, String, Integer"}$$

$$y = \varphi(\text{School})$$

$$x \xrightleftharpoons[\text{Method}]{\text{Class}} y$$

Relationship Mapping

Definition D.4 (Relationship Mapping) The set of meta-objects that maps a relationship specification $\sigma = (lc, lr, rc, rr, l, u, e, k)$, denoted as $Meta_R(\sigma)$, is the set containing only the meta-objects $r \in \xi(\text{Relationship})$, $x \in \xi(\text{Class})$, $y \in \xi(\text{Class})$, such that:

Meta_R

- (i) if $\sigma.e \in \{\text{PTAE}, \text{STAE}\}$ then $r.c = \text{TightAggregation}$

- (ii) if $\sigma.e \in \{\text{PLAE}, \text{SLAE}\}$ then $r.c = \text{LooseAggregation}$
 - (iii) if $\sigma.e = \text{ASSE}$ then $r.c = \text{Association}$
 - (iv) $x \xrightarrow[\text{LeftClass}]{\text{LeftRelationship}} r$
 - (v) $y \xrightarrow[\text{RightClass}]{\text{RightRelationship}} r$
 - (vi) either $((x = \varphi(\sigma.lc)) \text{ and } (y = \varphi(\sigma.rc)))$
or $((y = \varphi(\sigma.lc)) \text{ and } (x = \varphi(\sigma.rc)))$
 - (vii) if $\sigma.e \in \{\text{PTAE}, \text{PLAE}\}$ then:
 - (a) $x = \varphi(\sigma.lc)$
 - (b) $r \dashrightarrow \text{left_key} = \sigma.k$
 - (c) $r \dashrightarrow \text{aggregate_role} = \sigma.lr$
 - (d) $r \dashrightarrow \text{component_min_card} = \sigma.l$
 - (e) $r \dashrightarrow \text{component_max_card} = \sigma.u$
 - (viii) if $\sigma.e \in \{\text{STAE}, \text{SLAE}\}$ then:
 - (a) $y = \varphi(\sigma.lc)$
 - (b) $r \dashrightarrow \text{right_key} = \sigma.k$
 - (c) $r \dashrightarrow \text{component_role} = \sigma.lr$
 - (ix) if $\sigma.e = \text{SLAE}$ then:
 - (a) $r \dashrightarrow \text{aggregate_min_card} = \sigma.l$
 - (b) $r \dashrightarrow \text{aggregate_max_card} = \sigma.u$
 - (x) if $\sigma.e = \text{ASSE}$ and $x = \varphi(\sigma.lc)$ then:
-

$$(a) \ r \dashrightarrow \text{left_key} = \sigma.k$$

$$(b) \ r \dashrightarrow \text{left_role} = \sigma.lr$$

$$(c) \ r \dashrightarrow \text{right_min_card} = \sigma.l$$

$$(d) \ r \dashrightarrow \text{right_max_card} = \sigma.u$$

(xi) if $\sigma.e = \text{ASSE}$ and $y = \varphi(\sigma.lc)$ then:

$$(a) \ r \dashrightarrow \text{right_key} = \sigma.k$$

$$(b) \ r \dashrightarrow \text{right_role} = \sigma.lr$$

$$(c) \ r \dashrightarrow \text{left_min_card} = \sigma.l$$

$$(d) \ r \dashrightarrow \text{left_max_card} = \sigma.u$$

□

Proposition D.1 (Complementary Relationship Specifications Mapping) Given two relationship specifications σ and λ , if σ and λ are complementary then $\text{Meta}_R(\sigma) = \text{Meta}_R(\lambda)$. □

Example D.4 Let us consider the association between classes *School* and *Person*. A pair of complementary relationship specifications σ and λ for that association is given by:

$$\sigma = (\text{School}, \text{School}, \text{Person}, \text{Student}, 0, n, \text{ASSE}, 0)$$

$$\lambda = (\text{Person}, \text{Student}, \text{School}, \text{School}, 0, 3, \text{ASSE}, 1)$$

Let us designate the class `School` as the `LeftClass` and the class `Person` as the `RightClass` in the association. Thus, $Meta_R(\sigma) = Meta_R(\lambda) = \{r, x, y\}$ such that:

$$\begin{aligned}
 r.c &= \text{Association} \\
 x &= \varphi(\sigma.lc) = \varphi(\text{School}) & y &= \varphi(\lambda.lc) = \varphi(\text{Person}) \\
 x &\xrightarrow[\text{LeftClass}]{\text{LeftRelationship}} r & y &\xrightarrow[\text{RightClass}]{\text{RightRelationship}} r \\
 r \dashrightarrow \text{left_key} &= \sigma.k = 0 & r \dashrightarrow \text{right_key} &= \lambda.k = 1 \\
 r \dashrightarrow \text{left_role} &= \sigma.lr = \text{School} & r \dashrightarrow \text{right_role} &= \lambda.lr = \text{Student} \\
 r \dashrightarrow \text{left_min_card} &= \lambda.l = 0 & r \dashrightarrow \text{right_min_card} &= \sigma.l = 0 \\
 r \dashrightarrow \text{left_max_card} &= \lambda.u = 3 & r \dashrightarrow \text{right_max_card} &= \sigma.u = n
 \end{aligned}$$

Class Mapping

Meta_C

Definition D.5 (Class Mapping) The set of meta-objects that maps a class β , denoted as $Meta_C(\beta)$, is the set given by:

$$Meta_C(\beta) = \left(\bigcup_{\alpha \in \beta.A} Meta_A(\alpha) \right) \cup \left(\bigcup_{\mu \in \beta.M} Meta_M(\mu) \right) \cup \left(\bigcup_{\sigma \in \beta.R} Meta_R(\sigma) \right)$$

Proposition D.2 (Class Path Mapping) Given a class $\beta \in \mathcal{C}$, $\forall \lambda \in \mathcal{C}$: if $\beta \leq_\tau \lambda$ then $\varphi(\lambda.n) \in Meta_C(\beta)$. \square

Invariant D.1 (Inheritance Mapping) $\forall b, d \in \mathcal{C}$: if $d.s = b.n$ then

$$\varphi(d.n) \xrightarrow[\text{SubClass}]{\text{SuperClass}} \varphi(b.n)$$

Example D.5 Let β denote the class `University` ($\beta = \kappa_{\text{University}}$), then:

$$\beta.A = \{\alpha_1, \alpha_2\} \qquad \beta.M = \{\mu\} \qquad \beta.R = \{\sigma\}$$

where:

$$\alpha_1 = (\text{name}, \text{String}, 1, \text{School})$$

$$\alpha_2 = (\text{acronym}, \text{String}, 0, \text{University})$$

$$\mu = (\text{School}, (\text{register_student}, \langle \text{Person}, \text{String} \rangle, \text{Integer}), f : S \rightarrow T, b)$$

$$\sigma = (\text{School}, \text{School}, \text{Person}, \text{Student}, 0, n, \text{ASSE}, 0)$$

Thus, from Definition D.5 (Class Mapping), we have that:

$$\text{Meta}_C(\beta) = \left(\text{Meta}_A(\alpha_1) \cup \text{Meta}_A(\alpha_2) \right) \cup \left(\text{Meta}_M(\mu) \right) \cup \left(\text{Meta}_R(\sigma) \right)$$

Now, let us verify that Proposition D.2 holds. Since class *School* is the only superclass of β we have that all is-a relations of β are given by:

$$\kappa_{\text{University}} \leq_{\tau} \kappa_{\text{School}}$$

$$\kappa_{\text{University}} \leq_{\tau} \kappa_{\text{University}}$$

From Definition D.2 (Attribute Mapping), Definition D.3 (Method Mapping) and Definition D.4 (Relationship Mapping), we have that:

$$\text{Meta}_A(\alpha_1) \supset \{\varphi(\text{School})\} \quad \text{Meta}_M(\mu) \supset \{\varphi(\text{School})\}$$

$$\text{Meta}_A(\alpha_2) \supset \{\varphi(\text{University})\} \quad \text{Meta}_R(\sigma) \supset \{\varphi(\text{School}), \varphi(\text{Person})\}$$

Hence:

$$\text{Meta}_C(\beta) \supset \{\varphi(\text{School}), \varphi(\text{University})\}$$

Moreover, since class *School* is direct superclass of β ($\beta.s = \text{School}$), according to Invariant D.1, we have that:

$$\varphi(\text{University}) \xrightleftharpoons[\text{SubClass}]{\text{SuperClass}} \varphi(\text{School})$$

Schema Mapping

Invariant D.2 (Schema Instance Name Distinction) $\forall x, y \in \xi(\text{Schema})$: if $x \dashrightarrow \text{name} = y \dashrightarrow \text{name}$ then $x = y$. \blacklozenge

Definition D.6 (Schema Meta-object) Given a schema $w \in \mathcal{W}$, the schema meta-object with respect to w , denoted as $\gamma(w)$, is $m \in \xi(\text{Schema})$ such that $m \dashrightarrow \text{name} = w.n$. \square

Example D.6 Let us consider the schema depicted in Figure 6.2. For simplicity of notation, let us denote the schema by w_3 , and let us suppose that its name is Academia ($w_3.n = \text{Academia}$). Thus, the notation $\gamma(w_3)$ denotes the instance m of the meta-class Schema such that $m \dashrightarrow \text{name} = \text{Academia}$. \blacklozenge

Definition D.7 (Schema Mapping) The set of meta-objects that maps a schema w , denoted as $\text{Meta}_W(w)$, is the set given by:

$$\text{Meta}_W(w) = \{\gamma(w)\} \cup \left(\bigcup_{\beta \in w.H.V} \text{Meta}_C(\beta) \right) \cup \left(\bigcup_{x \in w.S} \text{Meta}_W(x) \right)$$

Invariant D.3 (Root-subtree Mapping) $\forall w \in \mathcal{W}$:

- (i) $\forall c \in w.H.V$: if $c = \text{Root}(w.H)$ then $\varphi(c.n) \xrightarrow[\text{RootClass}]{\text{RootSchema}} \gamma(w)$
- (ii) $\forall c \in w.H.V$: if $c \neq \text{Root}(w.H)$ then $\varphi(c.n) \xrightarrow[\text{NonRootClass}]{\text{NonRootSchema}} \gamma(w)$ \blacklozenge

Example D.7 Let us consider the schema depicted in Figure 6.2. Since there are two root classes (School and Person) the schema is a super-schema composed two basic schemas. For simplicity of notation, let us denote the basic schema rooted at class School by w_1 and the basic schema rooted at class Person by w_2 . Thus,

we have that:

$$\begin{array}{cc} \varphi(\text{School}) \xrightleftharpoons[\text{RootClass}]{\text{RootSchema}} \gamma(w_1) & \varphi(\text{Person}) \xrightleftharpoons[\text{RootClass}]{\text{RootSchema}} \gamma(w_2) \\ \varphi(\text{University}) \xrightleftharpoons[\text{NonRootClass}]{\text{NonRootSchema}} \gamma(w_1) & \end{array}$$

Invariant D.4 (Schema Nesting Mapping) $\forall w, s \in \mathcal{W}$: if $s \in w.S$ then:

$$\gamma(s) \xrightleftharpoons[\text{SubSchema}]{\text{SuperSchema}} \gamma(w)$$

Example D.8 Let us consider the super-schema depicted in Figure 6.2. Let us denote the sub-schema rooted at class *School* by w_1 , the sub-schema rooted at class *Person* by w_2 , and the super-schema by w_3 . Thus, we have that:

$$\begin{array}{cc} \gamma(w_1) \xrightleftharpoons[\text{SubSchema}]{\text{SuperSchema}} \gamma(w_3) & \gamma(w_2) \xrightleftharpoons[\text{SubSchema}]{\text{SuperSchema}} \gamma(w_3) \end{array}$$

Summary

We summarise the discussion on meta-object mapping by presenting, as an example, the complete set of meta-objects that maps the schema depicted in Figure 6.2. Formally, the schema elements are given as follows.

$$a_1 = (\text{name}, \text{String}, 1, \text{School})$$

$$a_2 = (\text{acronym}, \text{String}, 0, \text{University})$$

$$a_3 = (\text{surname}, \text{String}, 1, \text{Person})$$

$$a_4 = (\text{firstname}, \text{String}, 0, \text{Person})$$

$$a_5 = (\text{age}, \text{Integer}, 0, \text{Person})$$

$$m_1 = (\text{School}, (\text{register_student}, \langle \text{Person}, \text{String} \rangle, \text{Integer}), f : S \rightarrow T, b)$$

$$r_1 = (\text{School}, \text{School}, \text{Person}, \text{Student}, 0, n, \text{ASSE}, 0)$$

$$r_2 = (\text{Person}, \text{Student}, \text{School}, \text{School}, 0, 3, \text{ASSE}, 1)$$

$$c_1 = (\text{School}, \emptyset, \langle \text{School} \rangle, \{a_1\}, \{r_1\}, \{m_1\}, \{a_1\}, \{r_1\}, \{m_1\})$$

$$c_2 = (\text{University}, \text{School}, \langle \text{School}, \text{University} \rangle, \{a_2\}, \emptyset, \emptyset, \{a_1, a_2\}, \{r_1\}, \{\Gamma_{\text{University}}(r_1)\})$$

$$c_3 = (\text{Person}, \emptyset, \langle \text{Person} \rangle, \{a_3, a_4, a_5\}, \{r_2\}, \emptyset, \{a_3, a_4, a_5\}, \{r_2\}, \emptyset)$$

$$w_1 = (\text{School}, H_1, \emptyset), H_1.V = \{c_1, c_2\}$$

$$w_2 = (\text{Person}, H_2, \emptyset), H_2.V = \{c_3\}$$

$$w_3 = (\text{Academia}, \text{nil}, \{w_1, w_2\})$$

The set of meta-objects that map all schema elements is diagrammatically represented in Figure 6.4. Formally, the mapping of all schema elements are given as follows. We recall that the notation Θ_i denotes the object o whose name is i ($o.n = i$).

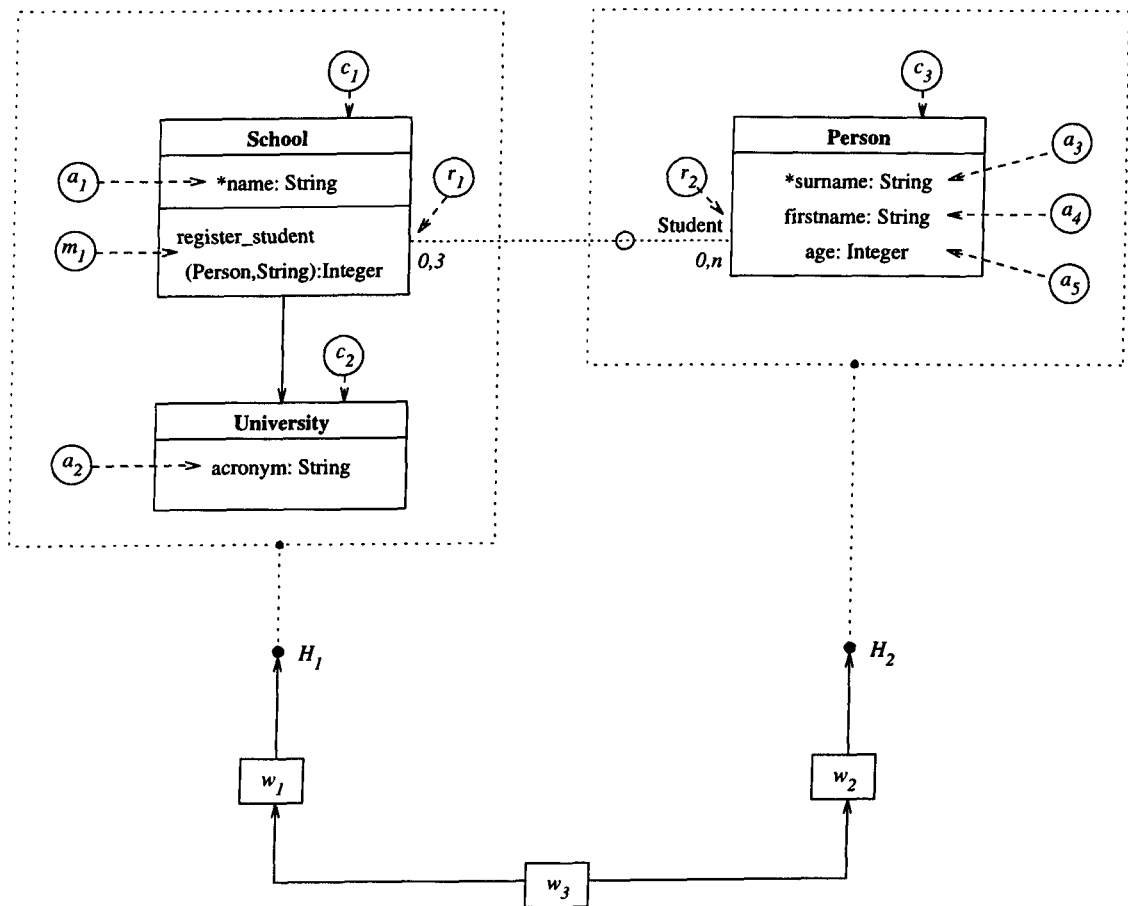


Figure D.1 Example schema with annotated formal elements

<p>Class Meta-objects</p> $\varphi(\text{School}) = \Theta_1$ $\varphi(\text{University}) = \Theta_4$ $\varphi(\text{Person}) = \Theta_6$	<p>Attribute Mapping</p> $Meta_A(a_1) = \{\Theta_2, \Theta_1\}, \Theta_2 \overset{\text{Class}}{\underset{\text{Attribute}}{=}} \Theta_1$ $Meta_A(a_2) = \{\Theta_5, \Theta_4\}, \Theta_5 \overset{\text{Class}}{\underset{\text{Attribute}}{=}} \Theta_4$ $Meta_A(a_3) = \{\Theta_7, \Theta_6\}, \Theta_7 \overset{\text{Class}}{\underset{\text{Attribute}}{=}} \Theta_6$ $Meta_A(a_4) = \{\Theta_8, \Theta_6\}, \Theta_8 \overset{\text{Class}}{\underset{\text{Attribute}}{=}} \Theta_6$ $Meta_A(a_5) = \{\Theta_9, \Theta_6\}, \Theta_9 \overset{\text{Class}}{\underset{\text{Attribute}}{=}} \Theta_6$
<p>Schema Meta-objects</p> $\gamma(w_1) = \Theta_{11}$ $\gamma(w_2) = \Theta_{12}$ $\gamma(w_3) = \Theta_{13}$	<p>Relationship Mapping</p> $Meta_R(r_1) = \{\Theta_{10}, \Theta_1, \Theta_6\}, \Theta_1 \overset{\text{LeftRelationship}}{\underset{\text{LeftClass}}{=}} \Theta_{10}$ $Meta_R(r_2) = \{\Theta_{10}, \Theta_1, \Theta_6\}, \Theta_6 \overset{\text{RightRelationship}}{\underset{\text{RightClass}}{=}} \Theta_{10}$
<p>Inheritance Mapping</p> $\Theta_4 \overset{\text{SuperClass}}{\underset{\text{SubClass}}{=}} \Theta_1$	<p>Method Mapping</p> $Meta_M(m_1) = \{\Theta_3, \Theta_1\}, \Theta_3 \overset{\text{Class}}{\underset{\text{Method}}{=}} \Theta_1$
<p>Class Mapping</p> $Meta_C(c_1) = Meta_A(a_1) \cup Meta_M(m_1) \cup Meta_R(r_1) = \{\Theta_1, \Theta_2, \Theta_3, \Theta_6, \Theta_{10}\}$ $Meta_C(c_2) = Meta_A(a_1) \cup Meta_A(a_2) \cup Meta_M(m_1) \cup Meta_R(r_1) = \{\Theta_1, \dots, \Theta_6, \Theta_{10}\}$ $Meta_C(c_3) = Meta_A(a_3) \cup Meta_A(a_4) \cup Meta_A(a_5) \cup Meta_R(r_2) = \{\Theta_1, \Theta_6, \dots, \Theta_{10}\}$	

Schema Mapping

$$Meta_W(w_1) = \{\gamma(w_1)\} \cup Meta_C(c_1) \cup Meta_C(c_2) = \{\Theta_1, \dots, \Theta_6, \Theta_{10}, \Theta_{11}\}$$

$$Meta_W(w_2) = \{\gamma(w_2)\} \cup Meta_C(c_3) = \{\Theta_1, \Theta_6, \dots, \Theta_{10}, \Theta_{12}\}$$

$$Meta_W(w_3) = \{\gamma(w_3)\} \cup Meta_W(w_1) \cup Meta_W(w_2) = \{\Theta_1, \dots, \Theta_{13}\}$$

Root-subtree Mapping

$$\begin{array}{ccc} \Theta_1 & \begin{array}{c} \text{RootSchema} \\ \rightleftharpoons \\ \text{RootClass} \end{array} & \Theta_{11} \\ \Theta_4 & \begin{array}{c} \text{NonRootSchema} \\ \rightleftharpoons \\ \text{NonRootClass} \end{array} & \Theta_{11} \\ \Theta_6 & \begin{array}{c} \text{RootSchema} \\ \rightleftharpoons \\ \text{RootClass} \end{array} & \Theta_{12} \end{array}$$

Schema Nesting Mapping

$$\begin{array}{ccc} \Theta_{11} & \begin{array}{c} \text{SuperSchema} \\ \rightleftharpoons \\ \text{SubSchema} \end{array} & \Theta_{13} \\ \Theta_{12} & \begin{array}{c} \text{SuperSchema} \\ \rightleftharpoons \\ \text{SubSchema} \end{array} & \Theta_{13} \end{array}$$

APPENDIX E

Components Definition

In this Appendix we formally define the index and organisational components of object engines, informally introduced in Chapter 3 and described in more detail in Chapter 7.

E.1 Indices

Attribute Indices

Definition E.1 (Object Attribute Index) Given an object $x \in \mathcal{O}$ and an attribute variable $\alpha \in x.A$ such that $\alpha.s.k = 1$, the object attribute index with respect to α , denoted as $Index_{OA}(x, \alpha)$, is a tuple (c, a, p, v, n) , where:

Index_{OA}

- $c \in CN$
- $a \in AN$
- $p \in PN$
- $v \in \mathcal{V}$
- $n \in ON$

such that:

- (i) $c = \alpha.s.c$
- (ii) $a = \alpha.s.n$
- (iii) $p = \alpha.s.p$
- (iv) $v = \alpha.v$
- (v) $n = x.n$ □

OAI *Notation E.1* The symbol *OAI* denotes the set of all object attribute indices. □

Invariant E.1 (Object Attribute Index Existence) $\forall x \in \mathcal{O} : \forall \alpha \in x.A \mid \alpha.s.k = 1 :$
 $\exists \iota \in OAI \mid \iota = Index_{OA}(x, \alpha).$ ♦

Relationship Indices

Index_{OR} *Definition E.2 (Object Relationship Index)* Given two objects $x, y \in \mathcal{O}$ and two complementary relationship variables $\sigma \in x.R$ and $\lambda \in y.R$ such that $\sigma.s.k = 1$, the object relationship index with respect to σ , denoted as $Index_{OR}(x, \sigma)$, is a tuple (lc, rc, rr, i, j) , where:

- $lc \in CN$
- $rc \in CN$
- $rr \in RN$
- $i \in ON$
- $j \in ON$

such that:

- (i) $lc = \sigma.s.lc$
- (ii) $rc = \sigma.s.rc$
- (iii) $rr = \sigma.s.rr$
- (iv) $i = x.n$
- (v) $j = y.n$
- (vi) $i \in \lambda.v$
- (vii) $j \in \sigma.v$ □

Notation E.2 The symbol *ORI* denotes the set of all object relationship indices. □ *ORI*

Invariant E.2 (Object Relationship Index Existence) $\forall x \in \mathcal{O} : \forall \sigma \in x.R \mid \sigma.s.k = 1 : \exists \iota \in ORI \mid \iota = Index_{OR}(x, \sigma)$. ◆

Class Indices

Definition E.3 (Class Attribute Index) Given a class $x \in \mathcal{C}$ and an attribute specification $s \in x.PA$ such that $s.k = 1$, the class attribute index with respect to s , denoted $Index_{CA}(x, s)$, is the set of object attribute indices given by:

$$Index_{CA}(x, s) = \{\iota \in OAI \mid \iota.c = x \wedge \iota.a = s.n\}$$

Definition E.4 (Class Attribute Indices) Given a class $x \in \mathcal{C}$, the attribute indices with respect to x , denoted $Index_A(x)$, is the set of object attribute indices given by:

$$Index_A(x) = \bigcup_{s \in x.PA} Index_{CA}(x, s)$$

*Index_{CA}**Index_A*

Index_{CR}

Definition E.5 (Class Relationship Index) Given a class $x \in \mathcal{C}$ and a relationship specification $s \in x.PR$ such that $s.k = 1$, the class relationship index with respect to s , denoted $Index_{CR}(x, s)$, is the set of object relationship indices given by:

$$Index_{CR}(x, s) = \{\iota \in ORI \mid \iota.lc = s.lc \wedge \iota.rr = s.rr\}$$

Index_R

Definition E.6 (Class Relationship Indices) Given a class $x \in \mathcal{C}$, the relationship indices with respect to x , denoted $Index_R(x)$, is the set of object relationship indices given by:

$$Index_R(x) = \bigcup_{s \in x.PR} Index_{CR}(x, s)$$

Index_C

Definition E.7 (Class Indices) Given a class $x \in \mathcal{C}$, the indices with respect to x , denoted $Index_C(x)$, is the set of object attribute and relationship indices given by:

$$Index_C(x) = Index_A(x) \cup Index_R(x)$$

Schema Indices

Index_W

Definition E.8 (Schema Indices) Given a schema $w \in \mathcal{W}$, the indices with respect to w , denoted as $Index_W(w)$, is the set of indices given by:

$$Index_W(w) = \bigcup_{x \in \Phi(w)} Index_C(x)$$

E.2 Views

Definition E.9 (View) Given a self-contained schema $w \in \mathcal{W}$, a view with respect to w is a tuple $(n, \delta, \Pi, \vartheta)$, where:

- $n \in \mathcal{R}_{WN}$
- δ is a database
- Π is a set of meta-objects
- ϑ is a set of indices

such that:

- (i) $n = w.n$
- (ii) $\delta = DB(w)$
- (iii) $\Pi = Meta_W(w)$
- (iv) $\vartheta = Index_W(w)$

□

Notation E.3 The symbol \mathcal{X} denotes the set of all views.

\mathcal{X}

Invariant E.3 (View Name Uniqueness) $\forall x, y \in \mathcal{X}$: if $x.n = y.n$ then $x = y$. ♦

Notation E.4 Given a schema w , the notation $View(w)$ denotes the view v such that $v.n = w.n$.

View

□

E.3 Contexts

Definition E.10 (Meta-view) Given a view $v \in \mathcal{X}$, if $v.n = \mathbf{Meta}$ then v is the meta-view in \mathcal{X} . □

Proposition E.1 (Meta-view and Meta-schema Correspondence) Given a view $v \in \mathcal{X}$ and a schema $w \in \mathcal{W}$, if v is the meta-view and w is the meta-schema then $v = View(w)$. □

Let us assume the existence of a countably infinite set ZN of context names.

Definition E.11 (Context) Given a set of object names ON , a context is a tuple $(n, PN, AN, MN, RN, CN, WN, \mathcal{D}, \mathcal{P}, \mathcal{C}, \mathcal{O}, \mathcal{I}, \mathcal{W}, \mathcal{X})$, where:

- $n \in ZN$
- PN is a set of primary type names
- AN is a set of attribute names
- MN is a set of method names
- RN is a set of role names
- CN is a set of class names
- WN is a set of schema names
- \mathcal{D} is a set of primary domains
- \mathcal{P} is a set of primary types
- \mathcal{C} is a set of classes
- \mathcal{O} is a set of objects
- \mathcal{I} is a set of indices
- \mathcal{W} is a set of schemas
- \mathcal{X} is a set of views

such that:

- (1) $\forall d \in \mathcal{D} : \exists t \in \mathcal{P}$ such that $d = \text{Dom}(t)$
 - (2) $\mathcal{D} \supseteq \mathcal{R}_{PD}$
 - (3) $\mathcal{P} \supseteq \mathcal{R}_{PT}$
-

$$(4) \quad PN \supseteq \mathcal{R}_{PN}$$

$$(5) \quad AN \supseteq \mathcal{R}_{AN}$$

$$(6) \quad CN \supseteq \mathcal{R}_{CN}$$

$$(7) \quad WN \supseteq \mathcal{R}_{WN}$$

$$(8) \quad \forall c \in \mathcal{C} :$$

$$(a) \quad c.n \in CN$$

$$(b) \quad c.s \in (CN \cup \{\emptyset\})$$

$$(c) \quad \forall a \in c.PA :$$

$$i. \quad a.s.n \in AN$$

$$ii. \quad a.s.p \in PN$$

$$(d) \quad \forall r \in c.PR :$$

$$i. \quad r.s.rc \in CN$$

$$ii. \quad r.s.lr \in RN$$

$$iii. \quad r.s.rr \in RN$$

$$(e) \quad \text{Let } TN = PN \cup CN, \text{ then } \forall m \in c.PM :$$

$$i. \quad m.s.n \in MN$$

$$ii. \quad \forall a \in m.s.A : a \in TN$$

$$iii. \quad m.s.r \in TN$$

$$(9) \quad \forall w \in \mathcal{W} :$$

$$(a) \quad w.n \in WN$$

$$(b) \quad \Phi(w) \subseteq \mathcal{C}$$

$$(10) \mathcal{O} = \bigcup_{c \in \mathcal{C}} \text{Ext}(c)$$

$$(11) \forall o \in \mathcal{O} : o.n \in ON$$

$$(12) \mathcal{I} = \bigcup_{x \in \mathcal{C}} \text{Index}_C(x)$$

$$(13) \forall w \in \mathcal{W} : \Phi(w) \subseteq \mathcal{C}$$

$$(14) \forall v \in \mathcal{X} : \exists w \in \mathcal{W} \text{ such that } \text{View}(w) = v$$

$$(15) \exists v \in \mathcal{X} \text{ such that } v.n = \text{Meta}$$

□

Invariant E.4 (Context Name Uniqueness) Let \mathcal{Z} be the set of all contexts, then
 $\forall x, y \in \mathcal{Z} : \text{if } x.n = y.n \text{ then } x = y.$ ♦

APPENDIX F

Query Language Syntax

The syntax for the query language is defined by the context-free grammar below. The grammar is specified by listing their productions. Each production defines a non-terminal symbol, called the *left side* of the production, through an Extended Backus-Naur Form (EBNF) expression, called the *right side* of the production. The start symbol of the syntax is the non-terminal defined by the first production, that is, the symbol *query*. The notation used is shown in the following Table:

Notation	Meaning
\rightarrow	Separation between left and right side
■	Termination of a production
	Separation of alternative right sides for the same left side
{ x }	A sequence of zero or more instances of x
$Not(x)$	Set complement in relation to x in a regular expression
\emptyset	Regular expression denoting the empty string
'xyz'	The terminal symbol xyz
xyz	The non-terminal symbol xyz

query $\rightarrow \emptyset \mid$ expression ■

expression \rightarrow class_expression \mid intersection_expression \mid
union_expression \mid '(' expression ')'

class_expression \rightarrow class_identifier where_clause ■

intersection_expression \rightarrow expression '&' expression ■

union_expression \rightarrow expression '|' expression ■

where_clause \rightarrow '(' where_expression ')'

where_expression $\rightarrow \emptyset \mid$ attribute_expression ■

attribute_expression \rightarrow term \mid and_expression \mid
or_expression \mid '(' attribute_expression ')'

term \rightarrow attribute_term \mid role_term ■

and_expression \rightarrow attribute_expression '&&' attribute_expression ■

or_expression \rightarrow attribute_expression '||' attribute_expression ■

attribute_term \rightarrow cast attribute_identifier relational_operator value ■

role_term \rightarrow cast role_identifier role_clause ■

cast $\rightarrow \emptyset \mid$ '[' class_identifier ']' ■

role_clause \rightarrow where_clause \mid '::' term ■

relational_operator \rightarrow '=' \mid '!=' \mid '>' \mid '<' \mid '>=' \mid '<=' \mid '%' ■

class_identifier \rightarrow identifier ■

attribute_identifier \rightarrow identifier ■

role_identifier \rightarrow identifier ■

identifier \rightarrow letter { letter | digit | '_' } ■

letter \rightarrow 'A' | 'B' | ... | 'Z' | 'a' | 'b' | ... | 'z' ■

digit \rightarrow '0' | '1' | ... | '9' ■

value \rightarrow string | integer ■

string \rightarrow ''' { string_element } ''' ■

string_element \rightarrow Not('') | '\ ' ■

integer \rightarrow unsigned_integer | sign unsigned_integer ■

unsigned_integer \rightarrow digit { digit } ■

sign \rightarrow '+' | '-' ■

Bibliography

- [1] S. Abiteboul, S. Cluet, and T. Milo. A database interface for file update. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pages 386–397, San Jose, California, USA, June 1995. ACM Press. SIGMOD RECORD, 24(2).
 - [2] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): the language and the data model. In *Proc. ACM SIGMOD*, Portland, OR, 1989.
 - [3] T. Andrews, C. Harris, and K. Sinkel. The Ontos Object Database, 1991.
 - [4] Architecture Projects Management Limited. *ANSA: An Engineer's Introduction to the Architecture*, Nov. 1989. Document TR.03.02.
 - [5] Architecture Projects Management Limited. *ANSAware 3.0 Implementation Manual*, Feb. 1991. Document RM.097.01.
 - [6] H. Assenmacher, T. Breitbach, P. Buhler, V. Hübsch, and R. Schwarz. PANDA - supporting distributed programming in C++. In O. M. Nierstrasz, editor, *Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 361–383, Kaiserslautern, Germany, July 1993. Springer-Verlag.
 - [7] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. Harvest: a scalable, customizable discovery and access system. Technical Report CU-CS-732-94, Department of Computer Science, University of Colorado, Boulder, Aug. 1994.
 - [8] C. M. Bowman, C. Dharap, M. Baruah, B. Camargo, and S. Potti. A file system for information management. In *Proceedings of the Conference on Intelligent Information Management Systems*, June 1994.
 - [9] M. Bowman, L. L. Peterson, and A. Yeatts. Univers: an attribute-based name server. *Software—Practice and Experience*, 20:403–424, Apr. 1990.
-

-
- [10] P. D. Bruza and van der Weide, T. P. Stratified hypermedia structures for information disclosure. *The Computer Journal*, 35(3):208–220, 1992.
 - [11] L. E. Buzato. *Management of Object-Oriented Action-Based Distributed Programs*. PhD thesis, University of Newcastle upon Tyne, Oct. 1994.
 - [12] L. E. Buzato and A. Calsavara. Stabilis: a case-study in writing fault-tolerant distributed applications using persistent objects. In *Proceedings of the 5th International Workshop on Persistent Object Systems*, San Miniato, Italy, Sept. 1992.
 - [13] Carnegie-Mellon University, Pittsburgh PA (USA). *Guide to the Camelot Distributed Transaction Facility: Release 1*, May 1988. A. Z. Spector and K. R. Swedlow, editors.
 - [14] J. S. Chase, F. G. Amador, and E. D. Lazowska. The Amber system: parallel programming on a network of multiprocessors. *Operating Systems Review*, 23(5):147–158, Dec. 1989. Proceedings of the 12th ACM Symposium on Operating Systems Principles.
 - [15] V. Christophides, S. Abiteboul, S. Cluet, and M. Scholl. From structured documents to novel query facilities. In SIGMOD94 [63], pages 313–324. SIGMOD RECORD, 23(2).
 - [16] M. P. Consens and T. Mile. Optimizing queries on files. In SIGMOD94 [63], pages 301–312. SIGMOD RECORD, 23(2).
 - [17] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. Simula 67 common base language. Technical Report S-22, Norwegian Computing Centre, Oslo, Oct. 1970.
 - [18] P. B. Danzig, S.-H. Li, and K. Obraczka. Distributed indexing of autonomous internet services. *Computing Systems*, 5(4):433–460, 1992.
 - [19] O. Deux et al. The story of O_2 . *IEEE Trans. on Data and Knowledge Engineering*, 2(1):91–108, Mar. 1989.
 - [20] O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):34–48, Oct. 1991.
 - [21] A. Emtage and P. Deutch. Archie — an electronic directory service for the Internet. In *Proceedings of the USENIX Winter Conference*, pages 93–110, Jan. 1992.
 - [22] D. K. Gifford, P. Jouvelot, M. A. Sheldon, and W. O’Toole Jr. Semantic File Systems. *Operating Systems Review*, 25(5), Oct. 1991. 13th ACM Symposium on Operating Systems.
-

-
- [23] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, Massachusetts, 1983.
 - [24] W. I. Grosky, F. Fotouhi, and I. K. Sethi. Using metadata for the intelligent browsing of structured media objects. *SIGMOD RECORD*, 23(4):49–56, Dec. 1994.
 - [25] D. Hardy and M. F. Schwartz. Essence: a resource discovery system based on semantic file indexing. In *Proceedings of the USENIX Winter Conference*, pages 361–374, Jan. 1993.
 - [26] D. J. Harper and A. D. M. Walker. ECLAIR: an extensible class library for information retrieval. *The computer journal*, 35(3):256–267, 1992.
 - [27] K. Harrentien, M. Stahl, and E. Feinler. Nicname/whois, Oct. 1985. RFC 954, SRI International.
 - [28] C. Hsu, M. Bouziane, L. Rattner, and L. Yee. Information resources management in heterogeneous distributed environments: a metadatabase approach. *IEEE Transactions on Software Engineering*, 17(6):604–625, June 1991.
 - [29] K. Järvelin and T. Niemi. An NF² relational interface for document retrieval, restructuring and aggregation. In *Proceedings of the 18th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 102–110. ACM Press, July 1995.
 - [30] B. Kahle and A. Medlar. An information system for corporate users: Wide Area Information Servers. *ConneXions - The Interoperability Report*, 5(11):2–9, Nov. 1991.
 - [31] S. N. Khoshafian and G. P. Copeland. Object identity. In *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages and Applications - OOPSLA'86*, pages 406–416, Portland, Oregon, Sept. 1986. ACM SIGPLAN Notices 21(11).
 - [32] W. Kim et al. Object-oriented concepts, databases and applications. In W. Kim and F. Lochovsky, editors, *Features of the ORION Object-Oriented Database System*, pages 251–282. Addison-Wesley, 1989.
 - [33] A. Kotz-Dittrich and K. R. Dittrich. Where object-oriented DBMSs should do better: A critique based on early experiences. In W. Kim, editor, *Modern Database Systems: The*
-

- Object Model, Interoperability, and Beyond*, chapter 12, pages 238–253. Addison-Wesley Publishing Company, 1995.
- [34] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–53, Oct. 1991.
- [35] L. Lamport. *L^AT_EX: User's Guide and Reference Manual*. Reading, Massachusetts, 1986. Appendix B: The Bibliography Database, pages 140–147.
- [36] B. Lampson. Designing a global name service. In *Proceedings of the Fifth Annual ACM Symposium on the Principles of Distributed Computing*, pages 1–10, Calgary, Aug. 1986. ACM, New York.
- [37] R. LeBlanc and C. T. Wilkes. Systems programming with objects and actions. In E. Swartzlander, editor, *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 132–139. Computer Society Press, Silver Spring, MD 20910, May 1985.
- [38] B. Liskov. Overview of the Argus language and system. Massachusetts Institute of Technology Laboratory for Computer Science, Programming Methodology Group Memo 40, Feb. 1984.
- [39] A. Loeffen. Text database: a survey of text models and systems. *SIGMOD RECORD*, 23(1):230–260, Mar. 1994.
- [40] C. A. Lynch. The Z39.50 information retrieval protocol: an overview and status report. *ACM Computer Communications Review*, 21(1):57–70, 1991.
- [41] M. S. Madsen, I. Fogg, and C. Ruggles. Metadata systems: Integrative information technologies. *Libri*, 44(3):237–257, 1994.
- [42] P. Mockapetris. Domain names - concepts and facilities. RFC-1034, USC-ISI.
- [43] P. Mockapetris. Domain names - implementation and specification. RFC-1035, USC-ISI.
- [44] B. C. Neuman. The Prospero file system: a global file system based on Virtual System Model. *Computing Systems*, 5(4):461–493, 1992.
- [45] Object Management Group. The common object request broker: Architecture and specification. OMG Document number 91.12.1, 1991. Revision 1.1.
-

-
- [46] K. Obraczka, P. B. Danzig, and S.-H. Li. Internet resource discovery services. *Computer*, pages 8–22, Sept. 1993.
- [47] Information processing - text and office systems - office document architecture (ODA) and interchange format, 1989.
- [48] Open Software Foundation. The OSF/1 operating systems. In *Proc. Spring EurOpen Conf.*, pages 33–42, Tromsø, Norway, May 1991.
- [49] J. Orenstein, S. Haradhvala, B. Margulies, and D. Sakahara. Query processing in the ObjectStore database system. In *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, USA, June 1992. ACM Press. SIGMOD RECORD, 21(2).
- [50] G. Parrington. Reliable distributed programming in C++: The Arjuna approach. In *Second Usenix C++ Conference*, pages 37 – 50, 1990.
- [51] G. D. Parrington, S. K. Shrivastava, S. M. Wheeler, and M. C. Little. The design and implementation of Arjuna. *USENIX Computing Systems Journal*, 8(3), 1995.
- [52] L. L. Peterson. The Profile naming service. *ACM Transactions on Computer Systems*, 6(4):341–364, Nov. 1988.
- [53] B. Potter, J. Sinclair, and D. Till. *An Introduction to Formal Specification and Z*. Prentice Hall, 1991.
- [54] J. Ranjit, M. Bowman, and M. Spasojevic. An extensible type system for wide-area information management. In *Proceedings of the 4th International Workshop on Object Orientation in Operating Systems*, pages 175–178, Lund, Sweden, Aug. 1995. IEEE Computer Society Press.
- [55] J. Rumbaugh, M. Blaha, W. Premierlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1991.
- [56] L. V. Saxton and V. Raghavan. Design of an integrated information retrieval-database management system. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):210–219, June 1990.
-

-
- [57] A. B. Schill and M. U. Mock. DC++: distributed object-oriented system support on top of OSF DCE. *Distributed Systems Engineering*, 1(2):112–125, 1993.
 - [58] M. F. Schwartz, B. Emtage, A. Kahle, and B. C. Neuman. A comparison of Internet resource discovery approaches. *Computing Systems*, 5(4):461–492, 1992.
 - [59] S. Sechrest and M. McClennen. Blending hierarchical and attribute-based file naming. In *Proceeding of the 12th International Conference on Distributed Computing Systems*, June 1992.
 - [60] Information Processing-Text and Office Systems-Standard Generalized Markup Language (SGML). Technical Report ISO 8879, ISO, 1986.
 - [61] S. Shrivastava and S. Wheeler. Implementing fault-tolerant distributed applications using objects and multi-coloured actions. In *Proceedings of the Tenth International Conference on Distributed Computing Systems*, pages 203 – 210, Paris, France, May 1990.
 - [62] S. K. Shrivastava, G. N. Dixon, and G. D. Parrington. An overview of the Arjuna programming system. *IEEE Software*, 8(1):66–73, Jan. 1991.
 - [63] *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, Minneapolis, Minnesota, USA, June 1994. ACM Press. SIGMOD RECORD, 23(2).
 - [64] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, 1986.
 - [65] D. B. Terry. Structure-free name management for evolving distributed environments. In *Proceedings of the 6th International Conference on Distributed Computing Systems*, pages 502–508, Cambridge, Massachusetts, May 1986. IEEE Computer Society Press.
 - [66] R. van der Linden. The ANSA naming model. Technical Report AR.003.01, APM Ltd., Cambridge, UK, 1993.
 - [67] R. J. Wilson. *Introduction to Graph Theory*. Longman Group Limited, third edition, 1985.
 - [68] X.500 Directory Service. CCITT Blue Book, 1988.
-